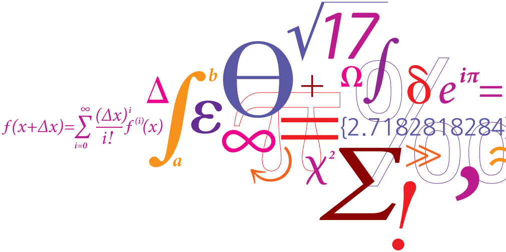
02157 Functional Programming

A brief introduction to Lambda calculus

Michael R. Hansen



DTU Informatics
Department of Informatics and Mathematical Modelling

Purpose

The theoretical underpinning of functional languages is λ-calculus.

The purpose is to hint on this underpinning and to introduce concepts of functional languages.

- Informal introduction to λ-calculus
    - computations of lambda-calculus and functional languages

Today you will be introduced to basic concepts of λ-calculus and you will get a feeling for the theoretical power of these concepts by the construction of an interpreter for a λ-calculus based language .

Lambda calculus: background

- Invented in the 1930's by the *logician* Alonzo Church in logical studies and in investigations of function definition and *application*, and *recursion*.
- Comprise full computability.
- First uncomputability results were discovered using λ-calculus.

Some questions

- Does the mathematical expression $x - y$ denote a function, say f, of x or a function, say g, of y, or ...?
- Does the notation $h(z)$ mean a function h or h applied to z

Lambda calculus: informal ideas

- $\lambda x.e$ denotes the *anonymous* function of x which e is.

Examples of *function definitions*:

- Let f be $\lambda x.x - y$
  The expression $x - y$ considered as a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ of x
- $g = \lambda y.x - y$
  The expression $x - y$ considered as a function $g : \mathbb{Z} \rightarrow \mathbb{Z}$ of y
- $h = \lambda x.\lambda y.x - y$
  The expression $x - y$ considered as a *higher-order* function
  $h : \mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z})$

Examples of *function applications*:

- $f(1) = 1 - y$ and $g(1) = x - 1$
- $h(2) = \lambda y.2 - y$ and $h(2)(5) = 2 - 5 = -3$

Lambda calculus: syntax

The set of *λ-terms or just terms* Λ is generated from a set V of variables by the rules:

- if $x \in V$, then $x \in \Lambda$ atom
- if $x \in V$ and $t \in \Lambda$, then $(\lambda x.t) \in \Lambda$ abstraction
- if $t_1, t_2 \in \Lambda$, then $(t_1 t_2) \in \Lambda$ application

Notational conventions to avoid brackets:

- Applications associated to the left, i.e. $t_1 t_2 t_3$ means $((t_1 t_2) t_3)$
- Abstraction extends as far as possible to the right, i.e. $\lambda x.PQ$ means $(\lambda x.(PQ))$
- $\lambda x_1 x_2 \cdots x_n.t$ means $(\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.t)\cdots)))$

Terms could be enriched with constants and constructs for pairs, e.g. $\lambda(x,y).x-y$ where $-$ is a constant.

Free and bound variables

A *occurrence* of a variable x is *bound* in t, if it occurs within the scope of an abstraction $\lambda x.M$ in t; otherwise it is *free*.

If x has at least one free occurrence in t, then it is called a free variable of t.

Examples:

- $x\,(\lambda y.y\,x)\,v$.
  Both occurrences of y are bound. Both x's are free. v is free.
- $(\lambda x.y\,x)(x\,z)$. Two left-hand occurrences of x are bound, the right-hand occurrence of x is free. x, y, z are free.

These concepts are also know from mathematics, logic and programming languages. For example:

- $\Sigma_{x=1}^{10} x^2$
- $\forall x.\exists y.x + y > 0 \Rightarrow z + x > y$
- `int f(int x,int y) = { return x + y - z; }`

where the occurrences of x and y are bound that those of z are free.

Substitution

Let $t[e/x]$ denote the term obtained from t by substituting e for every free occurrence of x.

In doing so, we rename bound variables to avoid clashes.

Examples:

- $(u\,\lambda x.y\,x)[v\,w/y] = u\,\lambda x.v\,w\,x$.
- $(y\,\lambda x.x\,y)[v\,w/y] = v\,w\lambda x.x\,(v\,w)$.
- $(\lambda x.y)[x/y] = (\lambda z.y)[x/y] = \lambda z.x$.
  Rename x to avoid clashes.

Comment:

- $(\lambda x.y)$ denote the constant function whose value is y.
- Therefore, $(\lambda x.y)[x/y]$ should intuitively denote the constant function with value x (as $\lambda z.x$ also does).
- The renaming is necessary as $\lambda x.x$ denotes the identity function.

α- conversions

Renaming bound variables does not change the meaning:

- $\Sigma_{x=1}^{10} x^2$ is equal to $\Sigma_{k=1}^{10} k^2$
- $\forall x.\exists y.x + y > 0 \Rightarrow z + x > y$ is equivalent to
  $\forall a.\exists b.a + b > 0 \Rightarrow z + a > b$
- `int f(int x,int y) = { return x + y - z; }`
  is the same as
  `int f(int a,int b) = { return a + b - z; }`

Renaming a bound variable in a term is called an α-conversion:

- Alpha: $\lambda x.t \;\rightarrow_\alpha\; \lambda y.t[y/x]$, when y is not free in t.
  renaming of bound variables

Example: $y(\lambda x.xz)y \;\rightarrow_\alpha\; y(\lambda v.vz)y$

β-reduction

The β-reduction rule formalizes the application of a function to an argument.

Example: $(\lambda x.x + 2)3$ reduces to 3+2.

- Beta: $(\lambda x.t)\,e \;\rightarrow_\beta\; t[e/x]$

function application

Examples

- $(\lambda x.\lambda y.x + y)\, a \;\to_\beta\; \lambda y.a + y$
- $\lambda x.\lambda y.\underline{(\lambda x.\lambda y.x\, x\, y)\, y}\, x \;\to_\beta\; \lambda x.\lambda y.\underline{(\lambda v.y\, y\, v)\, x} \;\to_\beta$ $\lambda x.\lambda y.y\, y\, x$
- $(\lambda x.a)\, b \;\to_\beta\; a$
- $\underline{(\lambda x.x)\,(\lambda y.a)}\, b \;\to_\beta\; (\lambda y.a)\, b \;\to_\beta\; a$

where free variables, e.g. $+, a, b$ are considered as constants.

Reductions may give bigger terms. Let $\Omega = \lambda x.x\, x\, x$.

$$\Omega\,\Omega \equiv \underline{(\lambda x.x\, x\, x)\,\Omega} \to_\beta \Omega\,\Omega\,\Omega \;\equiv \underline{(\lambda x.x\, x\, x)\,\Omega}\,\Omega \to_\beta \Omega\,\Omega\,\Omega\,\Omega \equiv \ldots$$

Termination depends on reduction strategy:

- $(\lambda x.a)\,(\Omega\,\Omega) \;\to_\beta^*\; (\lambda x.a)\,(\Omega\,\Omega\,\Omega) \;\to_\beta{}^*\; \ldots$
- $(\lambda x.a)\,(\Omega\,\Omega) \;\to_\beta\; a$

Lambda terms as programs

- A program p is a lambda term
- Computations are given by beta-reductions

In a ”real” functional programming language, the syntax is ”sugared” lambda terms, and computations are based on a specific strategy for applying beta-reduction.

In F# the reduction strategy is called *eager*. An application $e_1\, e_2$ is evaluated as follows:

- Evaluate e_1 to an abstraction $\lambda x.e$ (written $e_1 \leadsto \lambda x.e$).
- Evaluate e_2 to a value v (written $e_2 \leadsto v$).
- Perform the beta-reduction $(\lambda x.e)\; v \rightarrow_\beta e[v/x]$.

Hence, the ”bigstep” evaluation is $e_1\, e_2 \;\leadsto\; e[v/x]$

This eager strategy is efficient when functions need their arguments.

In the textbook the notion environment is used instead of substitution:

$$e_1\, e_2 \;\leadsto\; (e, [x \mapsto v]).$$

Church numerals. Natural number computations

Natural numbers are represented by, for example, *Church numerals*:

0	1	2	3	$\cdots$	n
$\lambda fx.x$	$\lambda f.\lambda x.fx$	$\lambda f.\lambda x.f(fx)$	$\lambda f.\lambda x.f(f(fx))$	$\ldots$	$\lambda f.\lambda x.f^n x$

where $f^0 x = x, f^{i+1}x = f^i(fx)$

- the main idea is to use a unary representation of numbers.
  Rather inefficient – but it works.

Let $\overline{n}$ denote the Church numeral for the natural number n.

Successor and additions operations

- Successor: $suc = \lambda n.\lambda f.\lambda x.n\,f\,(f\,x)$
- Addition: $add = \lambda m.\lambda n.\lambda f.\lambda x.m\,f\,(n\,f\,x)$

Reductions:

$$\begin{aligned}
\underline{suc\ \overline{n}} &= \lambda f.\lambda x.\underline{(\lambda f.\lambda x.f^{n}\,x)\,f}\,(f\,x)\\
&= \lambda f.\lambda x.\underline{(\lambda x.f^{n}\,x)\,(f\,x)}\\
&= \lambda f.\lambda x.f^{n}\,(f\,x) \;=\; \lambda f.\lambda x.f^{n+1}\,x \;=\; \overline{n+1}
\end{aligned}$$

$$\begin{aligned}
\underline{add\ \overline{m}\,\overline{n}} &= \lambda f.\lambda x.\underline{(\lambda f.\lambda x.f^{m}\,x)\,f}\,(\overline{n}\,f\,x)\\
&= \lambda f.\lambda x.\underline{(\lambda x.f^{m}\,x)\,(\overline{n}\,f\,x)}\\
&= \lambda f.\lambda x.f^{m}\,(\underline{(\lambda f.\lambda x.f^{n}\,x)\,f\,x})\\
&= \lambda f.\lambda x.f^{m}\,(f^{n}\,x) \;=\; \lambda f.\lambda x.f^{m+n}\,x \;=\; \overline{m+n}
\end{aligned}$$

Recursion in Lambda Calculus

How to make recursive functions in Lambda calculus?

Answer: use a *fixpoint combinator Y*.

- An element $x \in A$ is a fixpoint of a function $f : A \to A$, if $x = f(x)$
- A fixpoint combinator is a higher-order function Y that computes the fixpoint of another function F, i.e. $Y\,F = F(Y\,F)$

Example: Let $F = \lambda f.\lambda n.\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * f(n-1)$.

The factorial function $n!$ is a fixpoint for F, as

$$n! \;=\; \lambda n.\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (n-1)! \;=\; F\,n!$$

Thus the factorial function fact is declared by $Y\,F$, and e.g.

$$\begin{aligned} \mathit{fact}\ 2 = Y\,F\,2 = &\quad (\lambda n.\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (Y\,F\,(n-1)))\,2 \\ = &\quad \mathit{if}\ 2 = 0\ \mathit{then}\ 1\ \mathit{else}\ 2 * (Y\,F\,1) = \cdots = 2 \end{aligned}$$

Fixpoint combinators

There are many lambda terms Y satisfying: $Y\,F = F(Y\,F)$.

The first is due to Curry:

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

The second is due to Turing:

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

An advantage of Y_t over Y_c is that $Y_t\,F = F(Y_t\,F)$ can be established by reductions only (an exercise), i.e. Y_t is preferable for computational use.

Notice: These operators cannot be represented by an F# fun-expression. They contain self-applications (of the form $t\ t$) and these are not well-typed in F#.

Summary

- Brief introduction to lambda calculus.
- Hint a the theoretical underpinning of functional languages. (F# is actually more directly related to typed lambda calculus).
- Hint at the general computability capability of lambda calculus.

Have fun with the construction of a λ-calculus interpreter.