VU Semantik von Programmiersprachen

Agata Ciabattoni
Institute für Computersprachen,
Theory and Logic group
(agata@logic.at)

$\boxed{\text{(A gentle) Introduction to } \lambda \text{ calculus}}$

Why shoud I study λ calculus?

Why shoud I study λ calculus?

1. Historical importance

In 1936

- Turing invented the Turing machines
- Church invented the lambda calculus

In 1937 Turing proved that the two models for computation are equivalent. This lead to the Church-Turing thesis

> “The functions computed by algorithms coincide with the class of functions computable by Turing machines”.

Why shoud I study λ calculus? – II

1. Historical importance
2. Useful for studying formal semantics of programming languages
   - very simple (3 formation rules, 2 operations!!!!!!!)
   - powerful
   - effective paradigm to study binding mechanisms, parameter passing, ...
   - easy to extend with features of interest
   - plays a similar role for PL research as Turing machines do for computability

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus" (Landin '66)

Why shoud I study λ calculus? – III

1. Historical importance
2. Useful for studying formal semantics of programming languages
3. It's at the core of functional programming languages (LISP, ML..)

The design of the imperative languages is based on the von Neumann architecture (they consist of a sequence of statements)

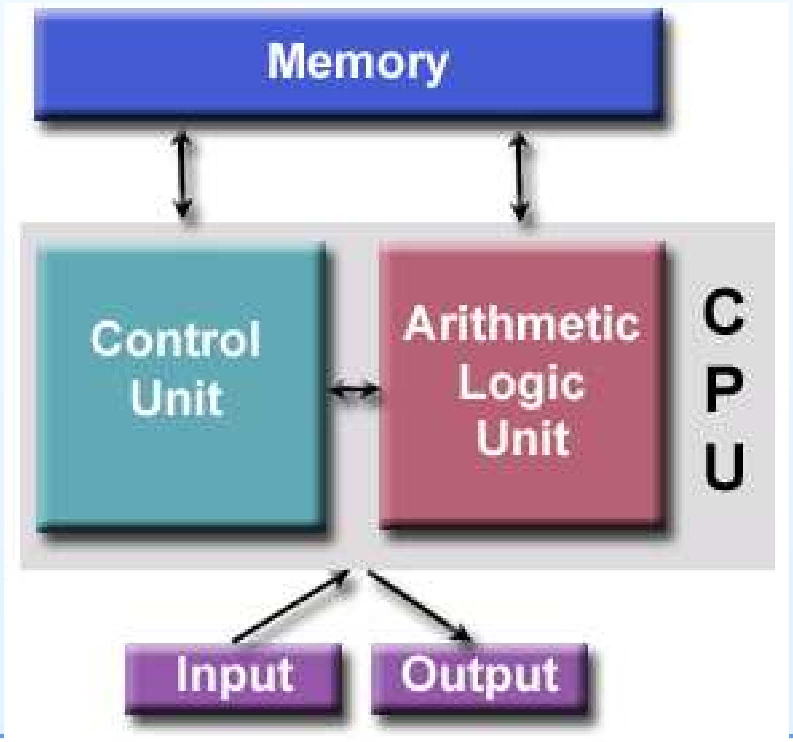
- Efficiency is the primary concern

The design of the functional languages is based on *mathematical functions*

- Solid theoretical basis but relatively unconcerned with the architecture

Recall: Von Neuman architecture

- One shared memory for instructions (program) and data
- One data bus and one address bus between processor and memory
- Instructions and data have to be fetched in sequential order (known as *Bottleneck*), limiting the operation bandwidth.



On functional programming languages

Functions are first-class objects. They can be passed as arguments to other functions and they can also be returned as results.

- “higher-order” way to program (functions can manipulate other functions)
- every expression denotes a single value
- the value cannot be changed by evaluating an expression or by sharing it between different parts of the program
- no references to global data: there is no global data
- no *side effects* (some impure functional programming languages do have them, e.g. Lisp, Scheme, ML)

Referential transparency: all expressions can be replaced with their value without changing the behavior of a program

The inventor of λ calculus: Alonso Church (1903-1995)

Had a few successful graduate students including

- Stephen Kleene (Regular expressions)
- Michael O. Rabin* (Nondeterministic automata)
- Dana Scott* (Formal programming language semantics)
- Alan Turing (Turing Machines)

$*$ Turing award winners

λ calculus

- mathematical formalism
- can be seen as a “core” (programming) language, easy and small but still Turing complete

λ notation

...blackboard explanations ...

Multi-argument functions

- Ex: adding two numbers 3 + 5, in λ notation

$$(\lambda x.\lambda y.x + y)3 \quad 5$$

- we partially evaluate λx, resulting in a new function

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

- this is equivalent to having a λ-binding with multiple arguments
- this is known as Currying

Syntax

λ-terms are built from a given, countable collection of

- variables $x, y, z \dots$

using two operations:

- λ abstraction: $(\lambda x.M)$
  (where x is a variable and M a λ-term)
- application: (MM')
  (where M and M' are λ-terms)

Variables

The variable in a term can be computed using the following algorithm

- varsOf x = $\{x\}$
- varsOf (M N) = varsOf M $\cup$ varsOf N
- varsOf $(\lambda x.M)$ = $\{x\}$ $\cup$ varsOf M

Note the form of this algorithm: a *structural induction*.

λ-Terms

Notational conventions:

- $(\lambda x_1 x_2 \ldots x_n . M)$ means $(\lambda x_1 .(\lambda x_2 . \ldots (\lambda x_n . M) \ldots)$
- $(M_1 M_2 \ldots M_n)$ means $(\ldots (M_1 M_2) \ldots M_n)$, i.e. application is left-associative
- drop outermost parentheses and those enclosing the body of a λ-abstraction

Scope of variables

- As in all languages with variables it is important to discuss the notion of scope

Recall: the scope of a variable (= identifier) is the portion of a program where the identifier is accessible

- An abstraction $\lambda x.M$ *binds* the variable x in M. We say that:
  - M is the *scope* of x
  - x is bound in $\lambda x.M$
  - just like formal function arguments are bound in the function body

Free and Bound Variables

In $\lambda x.M$ we call x the bound variable and M the body of the λ abstraction.

Free and Bound Variables

In $\lambda x.M$ we call x the bound variable and M the body of the λ abstraction.

An occurrence of x in a λ-term M is called

- binding, if in between λ and .
- bound if in the body of a binding occurrence of x
- free if neither binding not bound

Free and Bound Variables – formal definition

Sets of free and bound variables

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

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

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

$$BV(x) = \emptyset$$

$$BV(\lambda x.M) = BV(M) \cup \{x\}$$

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

If $FV(M) = \emptyset$, M is called a closed term, or combinator

Free and Bound Variables II

Static nested scope: the scope of an identifier is fixed at compile-time to be the smallest block (begin/end, function, or procedure body) containing the identifier's declaration.

Free and Bound Variables II

Static nested scope: the scope of an identifier is fixed at compile-time to be the smallest block (begin/end, function, or procedure body) containing the identifier's declaration.

Just like in any language with statically nested scoping we have to worry about variable shadowing (= a variable declared within a certain scope has the same name as a variable declared in an outer scope)

- An occurrence of a variable might refer to different things in different contexts

E.g.

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

Renaming bound variables

λ-terms that can be obtained from one another by renaming the bound variables are considered identical. This is called α-*equivalence*.

- E.g. $\lambda x.x$ is equivalent to $\lambda y.y$
- Intuition: by changing the name of a formal argument and of all its occurrences in the function body, the behaviour of the function does not change.

Try to rename bound variables so that they are unique: this makes it easy to see the scope of bindings and also prevent confusion!

α-Equivalence $M =_\alpha M'$

$\lambda x.M$ is intended to represent the function f such that

$$f(x) = M, \quad \text{for all } x.$$

α-Equivalence $M =_\alpha M'$

$\lambda x.M$ is intended to represent the function f such that

$$f(x) = M, \quad \text{for all } x.$$

Hence if $M' = M[x'/x]$ is the result of taking M and changing all occurrences of x to some variable x' that *does not appear free in* M, then $\lambda x.M$ and $\lambda x'.M'$ both represent the same function.

$$\lambda x.M \rightarrow^\alpha \lambda x'.M[x'/x]$$

λ calculus

Operational Semantics

β-Reduction

The process of evaluating lambda terms by "plugging arguments into functions".

$\lambda x.M$ can be seen as a function on λ-terms via substitution: map each N to $M[N/x]$.

β-Reduction

The process of evaluating lambda terms by "plugging arguments into functions".

$\lambda x.M$ can be seen as a function on λ-terms via substitution: map each N to $M[N/x]$.

So the natural notion of computation for λ-terms is given by stepping from a

β-redex $(\lambda x.M)N$

to the corresponding

β-reduct $M[N/x]$

Substitution $M[N/x]$

Watch out! We must be careful if the term we are substituting into has a λ inside.

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

$y[N/x] = y$ (if $x \neq y$)

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

$\lambda y.M[N/x] = \lambda y.(M[N/x])$ if $x \neq y$ and $y \notin FV(N)$

The side condition makes substitution "capture avoiding".

$\lambda y.M[N/x] = \lambda y'.(M\{^{y'}/_{y}\}[N/x])$ $\quad y'$ fresh

Evaluation and the static scope

- the definition of substitution (β-reduction) guarantees that evaluation respects stating scoping:

$$(\lambda x.(\lambda y.yx))(y(\lambda x.x)) \to \lambda z.z(y(\lambda v.v))$$

  (y remains free)

- if we forget to rename the bound y

$$(\lambda x.(\lambda y.yx))(y(\lambda x.x)) \to \lambda y.y(y(\lambda v.v))$$

  (y was free before but it is "captured" i.e. it is bound now)

Structural Operational Semantics

Single-step β-reduction: Is the smallest relation $\rightarrow_\beta$ on terms satisfying

- $$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]}\ (\beta)$$
- $$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N'}\ (cong_1) \qquad \frac{N \rightarrow_\beta N'}{MN \rightarrow_\beta M'N'}\ (cong_2)$$
- $$\frac{M \rightarrow_\beta N}{\lambda x.M \rightarrow_\beta \lambda x.N}\ (\eta)$$

(non deterministic semantics)

On reduction order

- Languages defined by non-deterministic sets of rules are common:
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
  - ...
- It is useful to know whether such system have the *diamond property*

β-reduction and β-equivalence

What it does mean for two λ terms to be computationally equivalent?

$\to\!\!\to$ is the reflexive and transitive closure of $\to^{\beta}$.
$M \to\!\!\to M'$ if M reduces to M' in zero or more steps.

β-reduction and β-equivalence

What it does mean for two λ terms to be computationally equivalent?

$\to\!\!\to$ is the reflexive and transitive closure of $\to^{\beta}$.
$M \to\!\!\to M'$ if M reduces to M' in zero or more steps.

$M =^{\beta} N$ holds if N can be obtained from M performing zero or more steps of $\alpha(+\eta)$-reductions, β-reductions and/or inverse reduction steps (i.e. making $\to_{\beta}$ symmetric).

Def.: $=^{\beta}$ is the reflexive, symmetric and transitive closure of $\to_{\beta}$, i.e. the smallest equivalence relation containing $\to_{\beta}$.

Reduction Order

The order in which you reduce things can matter.

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

Two things can be reduced:

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

... with different outcomes!

Normal Form

A lambda term that cannot be β-reduced is in *normal form*.

Normal Form

A lambda term that cannot be β-reduced is in *normal form*.

1. Can every λ term be reduced to a normal form?
2. If there is more than one reduction strategy, does each one lead to the same normal form?
3. Is there a reduction strategy that will guarantee that a normal form will be produced (if any)?

Non-termination (question 1)

Some λ terms have no β-nf

Ex.

$$\Omega := (\lambda x.xx)(\lambda x.xx)$$

Diamond Property

Church-Rosser Theorem

$\twoheadrightarrow$ is confluent, that is, if

$$M_1 \twoheadleftarrow M \twoheadrightarrow M_2$$

then there exists M' such that

$$M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$$

Normal Form (question 2)

Def.
A λ-term N is in β-normal form (β nf) if it contains no β-redex (i.e. no sub-terms of the form $(\lambda x.M)M')$.
M has β-nf N if $M =^{\beta} N$ with N a β-nf.

Normal Form (question 2)

Def.
A λ-term N is in β-normal form (β nf) if it contains no β-redex (i.e. no sub-terms of the form $(\lambda x.M)M'$).
M has β-nf N if $M =^{\beta} N$ with N a β-nf.

Note that if N is in β-nf and $N \to\!\!\to N'$, then $N =^{\alpha} N'$.

Normal Form (question 2)

Def.
A λ-term N is in β-normal form (β nf) if it contains no β-redex (i.e. no sub-terms of the form $(\lambda x.M)M'$).
M has β-nf N if $M =^\beta N$ with N a β-nf.

Note that if N is in β-nf and $N \rightarrow\!\!\!\rightarrow N'$, then $N =^\alpha N'$.

Corollary: The β-nf of M is unique up to α-equivalence, if it exists.
...proof on the blackboard

Evaluation Strategies

set of deterministic rules for evaluating expressions in a programming language

- (for functions) define when and in what order the arguments of a function are evaluated, when they are substituted into the function, and what form that substitution takes.
- λ calculus very often used to model evaluation strategies (reduction strategies).
- two basic groups: *strict* and *non-strict*, based on how function arguments are handled.

Evaluation Strategies – λ calculus

- Church Rosser Theorem says that independent of the reduction strategy we will not find more than one normal form
- But some reduction strategies might fail to find a normal form (question 3)
- We will consider the strategies:
  - normal order
  - applicative order

Def. A β-redex is *outermost* if there is no β-redex outside it and it is *innermost* if there is no β-redex inside it.

Applicative order

- reduce the leftmost innermost redex first
- always fully evaluate the arguments of a λ term before evaluating the term itself
- Call-by-value: functions are only called on *values* (= variable or lambda abstraction)

Normal-order

- reduce the leftmost outermost redex.
- Call-by-name: the arguments to a function are not evaluated before the function is called and rather, they are substituted directly into the function body and then left to be evaluated whenever they appear in the function.

Normal-order

- reduce the leftmost outermost redex.
- Call-by-name: the arguments to a function are not evaluated before the function is called and rather, they are substituted directly into the function body and then left to be evaluated whenever they appear in the function.

Not all λ terms have normal forms, but this a reliable way to find the normal form, if it exists.

Church Rosser Theorem II: If $M \to N$ and N is in normal form then there exists a *normal order* reduction sequence from M to N.

Evaluation strategies considerations

call-by-name

- more difficult to implement (must pass unvaluated expressions)
- the order of evaluation is harder to predict
- terminates more often than the call-by-value

call-by-value

- easy to implement
- well-behaved (predictable) w.r.t. side effects

Which is better?

λ calculus

Denotational Semantics: give meaning to a language by interpreting its terms as mathematical objects.

- function (*interpretation*) from terms to semantic objects

λ calculus

Denotational Semantics: give meaning to a language by interpreting its terms as mathematical objects.

- function (*interpretation*) from terms to semantic objects

We need a space D such that D is isomorphic to D^D (?)

D. Scott ('69) solved this problem by restricting D^D to the continous functions on D having a certain topology.

Computing with λ calculus

λ calculus can be seen as a “core” (programming) language, easy and small but still Turing complete

How can we possibly compute with the λ calculus when we have no data to manipulate?

1. no numbers
2. no data-structures
3. no control structures (if-then-else, loops)

Answer:

Use what we have to build these from scratch!

Encoding data in λ calculus

Computation in λ calculus is given by β-reduction. To relate this to computation in a programming language, or computation in other models we have to see how to encode numbers, pairs, lists... as λ-terms.

True and False

$$\mathbf{T} := \lambda xy.x \qquad \mathbf{F} := \lambda xy.y$$

Corollary $\mathbf{T} \neq \mathbf{F}$ (Church-Rosser theorem!)

Encoding data in λ calculus

Computation in λ calculus is given by β-reduction. To relate this to computation in a programming language, or computation in other models we have to see how to encode numbers, pairs, lists... as λ-terms.

True and False

$$\mathbf{T} := \lambda xy.x \qquad \mathbf{F} := \lambda xy.y$$

Corollary $\mathbf{T} \neq \mathbf{F}$ (Church-Rosser theorem!)

Examples of boolean functions

- **and**$:= \lambda ab.ab\mathbf{F}$
- **if-then-else**$:= \lambda a.a$

λ-definable functions

$f : \mathcal{N}^n \to \mathcal{N}$ is λ-definable if there is a closed λ-term F that represents it: for all $(x_1, \dots x_n) \in \mathcal{N}^n$ and $y \in \mathcal{N}$

- if $f(x_1, \dots x_n) = y$ then $F\underline{x_1} \dots \underline{x_n} =^\beta \underline{y}$
- if $f(x_1, \dots x_n) \uparrow$, then $F\underline{x_1} \dots \underline{x_n}$ has no (head) normal form

(notation: $\underline{\text{p}}$ stands for the encoding of the number p in λ calculus)

Computable = λ-definable

- Turing machines
- λ-calculus
- Recursive functions
- While language
- ...

Church’s numerals

$$\underline{n} := \lambda xy.\, \underbrace{x(x(\ldots(x}_{n\ times}\, y)\ldots)$$

Example

$$\begin{array}{rcl} \underline{0} & = & \lambda xy.y \\ \underline{1} & = & \lambda xy.xy \\ \underline{2} & = & \lambda xy.x(xy) \end{array}$$

$$(\underline{n}M)N =^{\beta} \underbrace{M(M(\ldots(M}_{n\ times}\, N)\ldots)$$

One example: Test$_0$ $:= \lambda npq.n(\mathbf{T}q)p$

Bibliography

Simple introduction http://krchowdhary.com/me-tfl/lambda-calculus-Chapter5.pdf

Selinger, Lecture Notes on the Lambda Calculus http://arxiv.org/pdf/0804.3434v1.pdf (Ch. 1, 2, 3.1, 3.2, 4.1, 4.2)

https://files.nyu.edu/cb125/public/Lambda/barendregt.94.pdf

Coding in λ calculus, http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf

Barendregt. Introduction to Lambda Calculus.