# Computer Science Course Notes — 1st Semester

Dario Loi

# Contents

# Models of Computation

## First Lesson

### Contact Information

Prof email: piperno@di.uniroma1.it

*Actual* lecture times:

- Wednsday 13:30 - 15:00
- Thursday 16:00 - 17:30, or 16:15 - 17:55

### Course Contents

The course content will focus on **functional programming** and $\lambda$-calculus.

The course contains practical exercises that should be done by pen and paper, so do not rely on *just* these notes.

The main application of these languages is that of *function application*, these languages are devoid of *assignment* semantics, and are therefore called *pure*.

The main ingredients of our programs will be:

- Variables, which are usually denoted by lowercase letters
- Starting from these, you obtain the set of all programs, which are called $\lambda$-terms, this set is denoted as $\Lambda$.

### Set Definition

We can provide an inductive definition of a set of $\lambda$-terms, $\Lambda$:

$$\frac{x \in V}{x \in \Lambda} \quad \text{(var)}$$

$$\frac{M \in \Lambda \quad N \in \Lambda}{(MN) \in \Lambda} \quad \text{(app)}$$

$$\frac{M \in \Lambda \quad x \in V}{\lambda x.M \in \Lambda} \quad \text{(abs)}$$

Applying these inductive rules (variables, application, abstraction).

## Bactus Normal Form

Another way of describing a set of lambda terms is by using Bactus Normal Form (BNF).

$$\Lambda :: Var|\Lambda\Lambda|\lambda Var.\Lambda$$

Which is essentially a grammar for the set of lambda terms.

Lambda calculus is left-associative:

$$((xy)z) = xyz \neq x(yz)$$

All functions are unary (we assume currying). For example, the function $f(x, y)$ is represented as:

$$\lambda x.\lambda y.fxy$$

.

Functions in lambda calculus can be applied to other functions or themselves, they can also return functions.

## Beta Reduction

The main operation in lambda calculus is *beta reduction*, which is the application of a function to an argument.

$$\underbrace{(\lambda x.M)N}_{redex} \rightarrow_\beta M[N/x]$$

Where $M[N/x]$ is the result of substituting all occurrences of $x$ in $M$ with $N$.

Some other examples:

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

$$(\lambda x.xx)y \rightarrow_\beta yy$$

$$(\lambda xy.yx)(\lambda u.u) \rightarrow_\beta \lambda y.y(\lambda u.u)$$

$$(\lambda xy.yx)(\lambda t.y) \rightarrow_\beta \lambda y.y(\lambda t.y)$$

This rule can be applied in any context in which it appears.

**Formal Substitution Definition**

We give a formal definition of substitution, $M[N/x]$:

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

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

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

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

As observed, substitution is always in place of *free* variables, therefore the abstraction is *not* replaced in the last rule.

If we had an abstraction of type $\lambda x.P$ where $x \in P$, it would be best to rename $x$ in order to avoid name clashes.

## Types of Variables

We distinguish two kinds of variables:

- Free variables: variables that are not bound by an abstraction
- Bound variables: variables that are bound by an abstraction

For example, in the term $\lambda x.xy$, $y$ is a free variable, while $x$ is a bound variable.

Bound variables can be renamed, whereas for free variables the naming is relevant.

$$\begin{cases} FV(x) & = \{x\} \\ FV(MN) & = FV(M) \cup FV(N) \\ FV(\lambda x.M) & = FV(M) - \{x\} \end{cases}$$

A set of lambda terms where $LM(\Lambda) =$ is called *closed*.

### Extra Rules

$$(\mu) \quad \frac{M \to_\beta M'}{NM \to_\beta NM'}$$

$$(\nu) \quad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$(\xi) \quad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

These rules allow us to select redexes in a context-free manner in the middle of our lambda term. We can then choose the order of evaluation of our redexes, while still taking care of the left-associative order of precedence. Our calculus is therefore not *determinate* but is still *deterministic*, meaning that there may be multiple reduction strategies but they all lead to the same result.

This corollary is called the *Church Rosser Theorem*, discovered in 1936.

In general, a call-by-value-like semantic is preferrable when choosing evaluation paths, as it clears the most amount of terms as early as possible.

- Call By Value is *efficient*
- Call By Name is *complete*, **if** the lambda term is normalizable

## Second Lecture

### Alpha Reduction

Alpha reduction is the renaming of bound variables in a lambda term.

$$\lambda x.M \to_\alpha \lambda y.M[y/x]$$

This rule is used to avoid name clashes between bound variables.

### Arithmetic Expressions

The set of all valid arithmetic expressions has a very precise syntax. In general, a syntax can be viewed either as a tool for checking validity or as a generator of valid expressions (a grammar).

We proceed to give a definition for arithmetic expressions

$$\frac{x \in \mathbb{N}}{x \in \text{Expr}} \quad \text{(num)}$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X + Y \in \text{Expr}} \quad \text{(add)}$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X \times Y \in \text{Expr}} \quad \text{(mul)}$$

Etc, etc… for all the other binary operations.

From this, we can successfully decompose any arithmetic expression into a syntactic tree. With this set of rules, we have a slight problem: we can't represent negative numbers. We could solve this either by adding a rule for unary minus, or by specifying the num rule over $\mathbb{Z}$ instead of $\mathbb{N}$.

## Combinators

We define three combinators:

$$S = \lambda xyz.xz(yz)$$

$$K = \lambda xy.x$$

$$I = \lambda u.u$$

We have that $SKy \to_\beta I$

*Exercise 1.1:* $\beta$-reduce $S(KS)S$

This reduces to $\lambda zbc.z(bc)$, which is the $B$ combinator (composition).

*Exercise 1.2:* $\beta$-reduce $S(BBS)(KK)$.

This reduces to $\lambda zcd.zdc$, which is the $C$ combinator (permutation).

For exercise this, we show a step-by-step reduction:

$$\begin{aligned}
&S(BBS)(KK)\\
\rightsquigarrow&\lambda z.(BBS)z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))zK\\
\rightsquigarrow&\lambda z.B(Sz)K\\
\rightsquigarrow&\lambda z.(\lambda c.(Sz)(Kc)\\
\rightsquigarrow&\lambda zc.Sz(Kc)\\
\rightsquigarrow&\lambda zcd.zd(Kcd)\\
\rightsquigarrow&\lambda zcd.zdc \quad \square.
\end{aligned}$$

## Third Lecture

`Recupera!!!`

## Fourth Lecture

First we do a simple exercise, $\beta$-reduce $\lambda uv.(\lambda z.zz)(\lambda t.tuv)$.

$$\begin{aligned}
&\lambda uv.(\lambda z.zz)(\lambda t.tuv)\\
\rightarrow_\beta &\lambda uv.(\lambda t.tuv)(\lambda t.tuv)\\
\rightarrow_\beta &(\lambda t.tuv)(\lambda t.tuv)\\
\rightarrow_\beta &\lambda uv.(\lambda t.tuv)uv\\
\rightarrow_\beta &\lambda uv.uuvv \quad \square.
\end{aligned}$$

Find a term $X$ s.t $Xx = \lambda t.t(Xx)$

$$\begin{aligned}
Xx &= \lambda t.t(Xx)\\
X &= (\lambda fxy.t(fx))X\\
X &= Y(\lambda fxy.t(fx))
\end{aligned}$$

Find a term $H$ s.t $H(\lambda x_1x_2x_3.P) = \lambda ax_3x_2x_1.ax_1x_2x_3$

## Fifth Lecture

Today we will try to be more precise about the ***fixed point*** operator.

The first question is:

Find $X$ s.t:

$$XMN = MI(MN) \quad \forall M, N \in \Lambda$$

By substitution, we have:

$$X = \lambda xy.xI(xy)$$

This can be verified by beta reduction (we skip it since it's trivial).

Now, the core question is to replicate the same result with an expression that is ***recursive*** in nature.

For example: find $X \in Lambda$ s.t:

$$XMN = M(XMN) \quad \forall M, N \in \Lambda$$

# Sixth Lecture

This lecture will focus on *Combinatory Logic*, which is a simpler version of lambda calculus, where we only use combinators.

Here we have: * Application * Variables * *No* Abstraction

This means that ***all*** variables are *free!*.

We have some ***constants*** (we define them using abstraction with a slight abuse of notation): * $S = \lambda xyz.xz(yz)$ * $K = \lambda xy.x$ * $I = \lambda x.x$ * $B = \lambda xyz.x(yz)$ * $C = \lambda xyz.xzy$

(Where $SK$ is a sufficient base for all combinators, and the other combinators are syntactic sugar).

We define the $CL$ set inductively, as follows:

$$\frac{x \in V}{x \in CL} \quad \text{(var)}$$

$$\frac{X \in const}{X \in CL} \quad \text{(const)}$$

$$\frac{X \in CL \quad Y \in CL}{(XY) \in CL} \quad \text{(app)}$$

In the same way as lambda calculus, we use left-associative application.

$$((UV)W) = UVW \neq U(VW)$$

Our constants have ***computational behavior***, as explained by the abstraction rules. We can give explicit definitions for constant computations in order to completely remove the abstraction rules.

- $SXYZ \triangleright XZ(YZ)$
- $KXY \triangleright X$
- $IX \triangleright X$
- $BXYZ \triangleright X(YZ)$
- $CXYZ \triangleright XZY$

Showing that these combinators map to lambda terms is trivial (we can just apply the definitions). Showing the opposite is a bit more complex, but it can be done.

We essentially want to ***implement*** the abstraction behavior from combinatory logic, so that we can transpose ***any*** lambda term (not just trivial combinators) into combinatory logic.

$$\lambda x.P \quad [x]P \quad \text{(abstraction of } x \text{ in } P)$$

For instance

$$\lambda xy.yx \quad [x]([y]yx)$$

## Transformation Algorithm

Our aim is to produce an algorithm that performs this transformation:

$$([x]P)x \to P$$

We do this by creating a set of rules that we then apply iteratively (effectively performing a closure of the initial expression over the transformation set).

We define indicators $U, V$ s.t:

$$\begin{cases} U & x \notin FV(U) \\ W, V & x \in FV(V) \end{cases}$$

Then we define the following rules:

$$\begin{aligned} \text{I.} \quad & [x]Ux = U \\ \text{II.} \quad & [x]x = I \\ \text{III.} \quad & [x]U = KU \\ \text{IV.} \quad & [x](UW) = BU([x]W) \\ \text{V.} \quad & [x][VU] = C([x]V)U \\ \text{VI.} \quad & [x](VW) = S([x]V)([x]W) \end{aligned}$$

The order of the rules is *important.* This kind of algorithm is called a *markov algorithm*, which is a class of algorithms in which a set of rules is applied iteratively, with a priority order, until a fixed point is reached.

## Example of application

Let's give an example of application by transforming

$$\lambda xy.ytx$$

We start by applying the rules:

$$
\begin{aligned}
&\lambda xy.ytx \\
&\quad [x]([y](yt)x) \\
&\quad [x](C([y]yt)x) \\
&\quad [x](C(C([y]y)t)x) \\
&\quad [x](C(C(I)t)x) \\
&\quad [x]C(CIt)x \\
&\quad C(CIt)x
\end{aligned}
$$

Now we can apply this combinatory logic expression to $xy$:

$$
\begin{aligned}
C(CIt)xy &\triangleright C(CIt)yx \\
&\triangleright CItyx \\
&\triangleright Iytx \\
&\triangleright ytx \quad \square.
\end{aligned}
$$

In order to show that our result is