# Introduction to Lambda Calculus

Henk Barendregt Erik Barendsen

Revised edition
December 1998, March 2000

# Chapter 1

# Introduction

### Some history

Leibniz had as ideal the following.

(1) *Create a 'universal language' in which all possible problems can be stated.*

(2) *Find a decision method to solve all the problems stated in the universal language.*

If one restricts oneself to mathematical problems, point (1) of Leibniz' ideal is fulfilled by taking some form of set theory formulated in the language of first order predicate logic. This was the situation after Frege and Russell (or Zermelo).

Point (2) of Leibniz' ideal became an important philosophical question. 'Can one solve all problems formulated in the universal language?' It seems not, but it is not clear how to prove that. This question became known as the *Entscheidungsproblem*.

In 1936 the *Entscheidungsproblem* was solved in the negative independently by Alonzo Church and Alan Turing. In order to do so, they needed a formalisation of the intuitive notion of 'decidable', or what is equivalent 'computable'. Church and Turing did this in two different ways by introducing two models of computation.

(1) Church (1936) invented a formal system called the *lambda calculus* and defined the notion of computable function via this system.

(2) Turing (1936/7) invented a class of machines (later to be called *Turing machines*) and defined the notion of computable function via these machines.

Also in 1936 Turing proved that both models are equally strong in the sense that they define the same class of computable functions (see Turing (1937)).

Based on the concept of a Turing machine are the present day *Von Neumann computers*. Conceptually these are Turing machines with random access registers. *Imperative programming languages* such as *Fortran*, *Pascal* etcetera as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements.

*Functional programming languages*, like *Miranda*, *ML* etcetera, are based on the lambda calculus. An early (although somewhat hybrid) example of such a language is *Lisp*. *Reduction machines* are specifically designed for the execution of these functional languages.

## Reduction and functional programming

A *functional program* consists of an expression $E$ (representing both the algorithm and the input). This expression $E$ is subject to some rewrite rules. Reduction consists of replacing a part $P$ of $E$ by another expression $P'$ according to the given rewrite rules. In schematic notation

$$E[P] \to E[P'],$$

provided that $P \to P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called *normal form* $E^*$ of the expression $E$ consists of the output of the given functional program.

An example:

$$\begin{array}{rcl} (7+4)*(8+5*3) & \to & 11*(8+5*3) \\ & \to & 11*(8+15) \\ & \to & 11*23 \\ & \to & 253. \end{array}$$

In this example the reduction rules consist of the 'tables' of addition and of multiplication on the numerals.

Also symbolic computations can be done by reduction. For example

first of (sort (append ('dog', 'rabbit') (sort (('mouse', 'cat'))))) →
→ first of (sort (append ('dog', 'rabbit') ('cat', 'mouse')))
→ first of (sort ('dog', 'rabbit', 'cat', 'mouse'))
→ first of ('cat', 'dog', 'mouse', 'rabbit')
→ 'cat'.

The necessary rewrite rules for append and sort can be programmed easily in a few lines. Functions like append given by some rewrite rules are called *combinators*.

Reduction systems usually satisfy the *Church-Rosser property*, which states that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced as follows:

$$\begin{array}{rcl} (7+4)*(8+5*3) & \to & (7+4)*(8+15) \\ & \to & 11*(8+15) \\ & \to & 11*23 \\ & \to & 253, \end{array}$$

or even by evaluating several expressions at the same time:

$$\begin{array}{rcl} (7+4)*(8+5*3) & \to & 11*(8+15) \\ & \to & 11*23 \\ & \to & 253. \end{array}$$

## Application and abstraction

The first basic operation of the $\lambda$-calculus is *application*. The expression

$$F \cdot A$$

or

$$FA$$

denotes the data $F$ considered as algorithm applied to the data $A$ considered as input. This can be viewed in two ways: either as the process of computation $FA$ or as the output of this process. The first view is captured by the notion of conversion and even better of reduction; the second by the notion of models (semantics).

The theory is *type-free*: it is allowed to consider expressions like $FF$, that is $F$ applied to itself. This will be useful to simulate recursion.

The other basic operation is *abstraction*. If $M \equiv M[x]$ is an expression containing ('depending on') $x$, then $\lambda x.M[x]$ denotes the function $x \mapsto M[x]$. Application and abstraction work together in the following intuitive formula.

$$(\lambda x.2 * x + 1)3 = 2 * 3 + 1 \quad (= 7).$$

That is, $(\lambda x.2 * x + 1)3$ denotes the function $x \mapsto 2 * x + 1$ applied to the argument 3 giving $2*3+1$ which is 7. In general we have $(\lambda x.M[x])N = M[N]$. This last equation is preferably written as

$$(\lambda x.M)N = M[x := N], \qquad (\beta)$$

where $[x := N]$ denotes substitution of $N$ for $x$. It is remarkable that although $(\beta)$ is the only essential axiom of the $\lambda$-calculus, the resulting theory is rather involved.

## Free and bound variables

Abstraction is said to *bind* the *free* variable $x$ in $M$. E.g. we say that $\lambda x.yx$ has $x$ as bound and $y$ as free variable. Substitution $[x := N]$ is only performed in the free occurrences of $x$:

$$yx(\lambda x.x)[x := N] \equiv yN(\lambda x.x).$$

In calculus there is a similar variable binding. In $\int_a^b f(x, y)dx$ the variable $x$ is bound and $y$ is free. It does not make sense to substitute 7 for $x$: $\int_a^b f(7, y)d7$; but substitution for $y$ makes sense: $\int_a^b f(x, 7)dx$.

For reasons of hygiene it will always be assumed that the bound variables that occur in a certain expression are different from the free ones. This can be fulfilled by renaming bound variables. E.g. $\lambda x.x$ becomes $\lambda y.y$. Indeed, these expressions act the same way:

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

and in fact they denote the same intended algorithm. Therefore expressions that differ only in the names of bound variables are identified.

## Functions of more arguments

Functions of several arguments can be obtained by iteration of application. The idea is due to Schönfinkel (1924) but is often called *currying*, after H.B. Curry who introduced it independently. Intuitively, if $f(x,y)$ depends on two arguments, one can define

$$\begin{aligned} F_x &= \lambda y.f(x,y), \\ F &= \lambda x.F_x. \end{aligned}$$

Then

$$(Fx)y = F_x y = f(x,y). \qquad (*)$$

This last equation shows that it is convenient to use *association to the left* for iterated application:

$$FM_1 \cdots M_n \text{ denotes } (\cdot\cdot((FM_1)M_2)\cdots M_n).$$

The equation $(*)$ then becomes

$$Fxy = f(x,y).$$

Dually, iterated abstraction uses *association to the right*:

$$\lambda x_1 \cdots x_n.f(x_1,\ldots,x_n) \text{ denotes } \lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.f(x_1,\ldots,x_n))\cdot\cdot)).$$

Then we have for $F$ defined above

$$F = \lambda xy.f(x,y)$$

and $(*)$ becomes

$$(\lambda xy.f(x,y))xy = f(x,y).$$

For $n$ arguments we have

$$(\lambda x_1 \cdots x_n.f(x_1,\cdots,x_n))x_1 \cdots x_n = f(x_1,\ldots,x_n)$$

by using $n$ times $(\beta)$. This last equation becomes in convenient vector notation

$$(\lambda\vec{x}.f[\vec{x}])\vec{x} = f[\vec{x}];$$

more generally one has

$$(\lambda\vec{x}.f[\vec{x}])\vec{N} = f[\vec{N}].$$

# Chapter 2

# Conversion

In this chapter, the $\lambda$-calculus will be introduced formally.

2.1. DEFINITION. The set of $\lambda$-*terms* (notation $\Lambda$) is built up from an infinite set of variables $V = \{v, v', v'', \ldots\}$ using application and (function) abstraction.

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \Lambda, \\ M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda, \\ M \in \Lambda, x \in V & \Rightarrow & (\lambda x M) \in \Lambda. \end{array}$$

In BN-form this is

$$\begin{array}{rcl} \mathsf{variable} & ::= & `v\text{'} \mid \mathsf{variable}\ `\,'\,\text{'} \\ \lambda\text{-}\mathsf{term} & ::= & \mathsf{variable} \mid `(\text{'}\ \lambda\text{-}\mathsf{term}\ \lambda\text{-}\mathsf{term}\ `)\text{'} \mid `(\lambda\text{'}\ \mathsf{variable}\ \lambda\text{-}\mathsf{term}\ `)\text{'} \end{array}$$

2.2. EXAMPLE. The following are $\lambda$-terms.

$$\begin{array}{c} v'; \\ (v'v); \\ (\lambda v(v'v)); \\ ((\lambda v(v'v))v''); \\ (((\lambda v(\lambda v'(v'v)))v'')v'''). \end{array}$$

2.3. CONVENTION. (i) $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary $\lambda$-terms. Outermost parentheses are not written.

(ii) $M \equiv N$ denotes that $M$ and $N$ are the same term or can be obtained from each other by renaming bound variables. E.g.

$$\begin{array}{rcl} (\lambda xy)z & \equiv & (\lambda xy)z; \\ (\lambda xx)z & \equiv & (\lambda yy)z; \\ (\lambda xx)z & \not\equiv & z; \\ (\lambda xx)z & \not\equiv & (\lambda xy)z. \end{array}$$

(iii) We use the abbreviations

$$FM_1 \cdots M_n \equiv (\cdot\cdot((FM_1)M_2) \cdots M_n)$$

and

$$\lambda x_1 \cdots x_n.M \equiv \lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))\cdot\cdot)).$$

The terms in Example 2.2 now may be written as follows.

$$y;$$
$$yx;$$
$$\lambda x.yx;$$
$$(\lambda x.yx)z;$$
$$(\lambda xy.yx)zw.$$

Note that $\lambda x.yx$ is $(\lambda x(yx))$ and not $((\lambda x.y)x)$.

2.4. Definition. (i) The set of *free variables* of $M$, notation $\mathrm{FV}(M)$, is defined inductively as follows.

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

A variable in $M$ is *bound* if it is not free. Note that a variable is bound if it occurs under the scope of a $\lambda$.

(ii) $M$ is a *closed $\lambda$-term* (or *combinator*) if $\mathrm{FV}(M)=\emptyset$. The set of closed $\lambda$-terms is denoted by $\Lambda^o$.

(iii) The result of *substituting* $N$ for the free occurences of $x$ in $M$, notation $M[x:=N]$, is defined as follows.

$$\begin{aligned}x[x:=N] &\equiv N;\\ y[x:=N] &\equiv y, \quad \text{if } x\not\equiv y;\\ (M_1M_2)[x:=N] &\equiv (M_1[x:=N])(M_2[x:=N]);\\ (\lambda y.M_1)[x:=N] &\equiv \lambda y.(M_1[x:=N]).\end{aligned}$$

2.5. Example. Consider the $\lambda$-term

$$\lambda xy.xyz.$$

Then $x$ and $y$ are bound variables and $z$ is a free variable. The term $\lambda xy.xxy$ is closed.

2.6. Variable convention. If $M_1,\ldots,M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Note that in the fourth clause of Definition 2.4 (iii) it is not needed to say 'provided that $y\not\equiv x$ and $y\notin\mathrm{FV}(N)$'. By the variable convention this is the case.

Now we can introduce the $\lambda$-calculus as formal theory.

2.7. DEFINITION. (i) The principal axiom scheme of the $\lambda$-calculus is

$$(\lambda x.M)N = M[x := N] \tag{$\beta$}$$

for all $M, N \in \Lambda$.

(ii) There are also the 'logical' axioms and rules.

Equality:

$$\begin{aligned} && M = M; \\ M = N &\Rightarrow& N = M; \\ M = N, N = L &\Rightarrow& M = L. \end{aligned}$$

Compatibility rules:

$$\begin{aligned} M = M' &\Rightarrow& MZ = M'Z; \\ M = M' &\Rightarrow& ZM = ZM'; \\ M = M' &\Rightarrow& \lambda x.M = \lambda x.M'. \end{aligned} \tag{$\xi$}$$

(iii) If $M = N$ is provable in the $\lambda$-calculus, then we sometimes write $\boldsymbol{\lambda} \vdash M = N$.

As a consequence of the compatibility rules, one can replace (sub)terms by equal terms in any term context:

$$M = N \Rightarrow \boxed{\cdots M \cdots} = \boxed{\cdots N \cdots}.$$

For example, $(\lambda y.yy)x = xx$, so

$$\boldsymbol{\lambda} \vdash \lambda x.x((\lambda y.yy)x)x = \lambda x.x(xx)x.$$

2.8. REMARK. We have identified terms that differ only in the names of bound variables. An alternative is to add to the $\lambda$-calculus the following axiom scheme

$$\lambda x.M = \lambda y.M[x := y], \quad \text{provided that } y \text{ does not occur in } M. \tag{$\alpha$}$$

We prefer our version of the theory in which the identifications are made on syntactic level. These identifications are done in our mind and not on paper. For implementations of the $\lambda$-calculus the machine has to deal with this so called $\alpha$-conversion. A good way of doing this is provided by the name-free notation of de Bruijn, see Barendregt (1984), Appendix C.

2.9. LEMMA. $\boldsymbol{\lambda} \vdash (\lambda x_1 \cdots x_n.M)X_1 \cdots X_n = M[x_1 := X_1] \cdots [x_n := X_n]$.

PROOF. By the axiom $(\beta)$ we have

$$(\lambda x_1.M)X_1 = M[x_1 := X_1].$$

By induction on $n$ the result follows. $\square$

2.10. EXAMPLE (Standard combinators). Define the combinators

$$\begin{aligned} \mathbf{I} &\equiv \lambda x.x; \\ \mathbf{K} &\equiv \lambda xy.x; \\ \mathbf{K}_* &\equiv \lambda xy.y; \\ \mathbf{S} &\equiv \lambda xyz.xz(yz). \end{aligned}$$

Then, by Lemma 2.9, we have the following equations.

$$\begin{aligned} \mathbf{I}M &= M; \\ \mathbf{K}MN &= M; \\ \mathbf{K}_*MN &= N; \\ \mathbf{S}MNL &= ML(NL). \end{aligned}$$

Now we can solve simple equations.

2.11. EXAMPLE. $\exists G\ \forall X\ GX = XXX$ (there exists $G \in \Lambda$ such that for all $X \in \Lambda$ one has $\boldsymbol{\lambda} \vdash GX = XX$). Indeed, take $G \equiv \lambda x.xxx$ and we are done.

Recursive equations require a special technique. The following result provides one way to represent recursion in the $\lambda$-calculus.

2.12. FIXEDPOINT THEOREM. (i) $\forall F\ \exists X\ FX = X$. (*This means: for all $F \in \Lambda$ there is an $X \in \Lambda$ such that $\boldsymbol{\lambda} \vdash FX = X$.*)

(ii) *There is a fixed point combinator*

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

*such that*

$$\forall F\ F(\mathbf{Y}F) = \mathbf{Y}F.$$

PROOF. (i) Define $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX.$$

(ii) By the proof of (i). □

2.13. EXAMPLE. (i) $\exists G\ \forall X\ GX = \mathbf{S}GX$. Indeed,

$$\begin{aligned} \forall X\ GX = \mathbf{S}GX &\Leftarrow Gx = \mathbf{S}Gx \\ &\Leftarrow G = \lambda x.\mathbf{S}Gx \\ &\Leftarrow G = (\lambda gx.\mathbf{S}gx)G \\ &\Leftarrow G \equiv \mathbf{Y}(\lambda gx.\mathbf{S}gx). \end{aligned}$$

Note that one can also take $G \equiv \mathbf{YS}$.

(ii) $\exists G\ \forall X\ GX = GG$: take $G \equiv \mathbf{Y}(\lambda gx.gg)$. (Can you solve this without using $\mathbf{Y}$?)

In the lambda calculus one can define numerals and represent numeric functions on them.

2.14. DEFINITION. (i) $F^n(M)$ with $F \in \Lambda$ and $n \in \mathbb{N}$ is defined inductively as follows.

$$\begin{aligned} F^0(M) &\equiv M; \\ F^{n+1}(M) &\equiv F(F^n(M)). \end{aligned}$$

(ii) The *Church numerals* $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$ are defined by

$$\boldsymbol{c}_n \equiv \lambda fx.f^n(x).$$

2.15. PROPOSITION (J.B. Rosser). *Define*

$$\begin{aligned} \mathsf{A}_+ &\equiv \lambda xypq.xp(ypq); \\ \mathsf{A}_* &\equiv \lambda xyz.x(yz); \\ \mathsf{A}_{\exp} &\equiv \lambda xy.yx. \end{aligned}$$

*Then one has for all* $n, m \in \mathbb{N}$

(i) $\mathsf{A}_+\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{n+m}$.
(ii) $\mathsf{A}_*\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{n*m}$.
(iii) $\mathsf{A}_{\exp}\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{(n^m)}$, *except for* $m = 0$ *(Rosser started counting from* $1$*).*

In the proof we need the following.

2.16. LEMMA. (i) $(\boldsymbol{c}_n x)^m(y) = x^{n*m}(y)$.
(ii) $(\boldsymbol{c}_n)^m(x) = \boldsymbol{c}_{(n^m)}(x)$, *for* $m > 0$.

PROOF. (i) Induction on $m$. If $m = 0$, then LHS $= y =$ RHS. Assume (i) is correct for $m$ (Induction Hypothesis: IH). Then

$$\begin{aligned} (\boldsymbol{c}_n x)^{m+1}(y) &= \boldsymbol{c}_n x((\boldsymbol{c}_n x)^m(y)) \\ &= \boldsymbol{c}_n x(x^{n*m}(y)) \quad \text{by IH,} \\ &= x^n(x^{n*m}(y)) \\ &\equiv x^{n+n*m}(y) \\ &\equiv x^{n*(m+1)}(y). \end{aligned}$$

(ii) Induction on $m > 0$. If $m = 1$, then LHS $\equiv \boldsymbol{c}_n x \equiv$ RHS. If (ii) is correct for $m$, then

$$\begin{aligned} (\boldsymbol{c}_n)^{m+1}(x) &= \boldsymbol{c}_n((\boldsymbol{c}_n)^m(x)) \\ &= \boldsymbol{c}_n(\boldsymbol{c}_{(n^m)}(x)) \quad \text{by IH,} \\ &= \lambda y.(\boldsymbol{c}_{(n^m)}(x))^n(y) \\ &= \lambda y.x^{n^m*n}(y) \quad \text{by (i),} \\ &= \boldsymbol{c}_{(n^{m+1})}x. \end{aligned}$$

PROOF OF THE PROPOSITION. (i) Exercise.

(ii) Exercise. Use Lemma 2.16 (i).

(iii) By Lemma 2.16 (ii) we have for $m > 0$

$$\begin{aligned} \mathsf{A}_{\exp}\boldsymbol{c}_n\boldsymbol{c}_m &= \boldsymbol{c}_m\boldsymbol{c}_n \\ &= \lambda x.(\boldsymbol{c}_n)^m(x) \\ &= \lambda x.\boldsymbol{c}_{(n^m)}x \\ &= \boldsymbol{c}_{(n^m)}, \end{aligned}$$

since $\lambda x.Mx = M$ if $M \equiv \lambda y.M'[y]$ and $x \notin \mathrm{FV}(M)$. Indeed,

$$\begin{aligned} \lambda x.Mx &\equiv \lambda x.(\lambda y.M'[y])x \\ &= \lambda x.M'[x] \\ &\equiv \lambda y.M'[y] \\ &\equiv M. \ \square \end{aligned}$$

## Exercises

2.1. (i) Rewrite according to official syntax

$$M_1 \equiv y(\lambda x.xy(\lambda zw.yz)).$$

(ii) Rewrite according to the simplified syntax

$$M_2 \equiv \lambda v'(\lambda v''((((\lambda vv)v')v'')((v''(\lambda v'''(v'v''')))v''))).$$

2.2. Prove the following *substitution lemma*. Let $x \not\equiv y$ and $x \notin \mathrm{FV}(L)$. Then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

2.3. (i) Prove, using Exercise 2.2,

$$\boldsymbol{\lambda} \vdash M_1 = M_2 \;\Rightarrow\; \boldsymbol{\lambda} \vdash M_1[x := N] = M_2[x := N].$$

(ii) Show

$$\boldsymbol{\lambda} \vdash M_1 = M_2 \,\&\, \boldsymbol{\lambda} \vdash N_1 = N_2 \;\Rightarrow\; \boldsymbol{\lambda} \vdash M_1[x := N_1] = M_2[x := N_2].$$

2.4. Prove Proposition 2.15 (i), (ii).

2.5. Let $\mathsf{B} \equiv \lambda xyz.x(yz)$. Simplify $M \equiv \mathsf{B}XYZ$, that is find a 'simple' term $N$ such that $\boldsymbol{\lambda} \vdash M = N$.

2.6. Simplify the following terms.
(i) $M \equiv (\lambda xyz.zyx)aa(\lambda pq.q)$;
(ii) $M \equiv (\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\mathsf{I})$;
(iii) $M \equiv \mathsf{SKSKSK}$.

2.7. Show that
(i) $\boldsymbol{\lambda} \vdash \mathsf{KI} = \mathsf{K}_*$;
(ii) $\boldsymbol{\lambda} \vdash \mathsf{SKK} = \mathsf{I}$.

2.8. (i) Write down a closed $\lambda$-term $F \in \Lambda$ such that for all $M, N \in \Lambda$

$$FMN = M(NM)N.$$

(ii) Construct a $\lambda$-term $F$ such that for all $M, N, L \in \Lambda^o$

$$FMNL = N(\lambda x.M)(\lambda yz.yLM).$$

2.9. Find closed terms $F$ such that
(i) $Fx = x\mathsf{I}$;
(ii) $Fxy = x\mathsf{I}y$.

2.10. Find closed terms $F$ such that
(i) $Fx = F$. This term can be called the 'eater' and is often denoted by $\mathsf{K}_\infty$;
(ii) $Fx = xF$;
(iii) $F\mathsf{IKK} = F\mathsf{K}$.

2.11. Show

$$\forall C[\ ,\ ]\ \exists F\ \forall \vec{x}\ F\vec{x} = C[F, \vec{x}]$$

and take another look at the exercises 2.8, 2.9 and 2.10.

2.12. Let $P, Q \in \Lambda$. $P$ and $Q$ are *incompatible*, notation $P \sharp Q$, if $\boldsymbol{\lambda}$ extended with $P = Q$ as axiom proves every equation between $\lambda$-terms, i.e. for all $M, N \in \Lambda$ one has $\boldsymbol{\lambda} + (P = Q) \vdash M = N$. In this case one says that $\boldsymbol{\lambda} + (P = Q)$ is *inconsistent*.
(i) Prove that for $P, Q \in \Lambda$

$$P \sharp Q \ \Leftrightarrow\ \boldsymbol{\lambda} + (P = Q) \vdash \mathsf{true} = \mathsf{false},$$

where $\mathsf{true} \equiv \mathsf{K}$, $\mathsf{false} \equiv \mathsf{K}_*$.
(ii) Show that $\mathsf{I} \sharp \mathsf{K}$.
(iii) Find a $\lambda$-term $F$ such that $F\mathsf{I} = x$ and $F\mathsf{K} = y$.
(iv) Show that $\mathsf{K} \sharp \mathsf{S}$.

2.13. Write down a grammar in BN-form that generates the $\lambda$-terms exactly in the way they are written in Convention 2.3.

# Chapter 4

# Reduction

There is a certain asymmetry in the basic scheme ($\beta$). The statement

$$(\lambda x.x^2 + 1)3 = 10$$

can be interpreted as '10 is the result of computing $(\lambda x.x^2 + 1)3$', but not vice versa. This computational aspect will be expressed by writing

$$(\lambda x.x^2 + 1)3 \twoheadrightarrow 10$$

which reads '$(\lambda x.x^2 + 1)3$ *reduces to* 10'.

Apart from this conceptual aspect, reduction is also useful for an analysis of convertibility. The Church-Rosser theorem says that if two terms are convertible, then there is a term to which they both reduce. In many cases the inconvertibility of two terms can be proved by showing that they do not reduce to a common term.

4.1. DEFINITION. (i) A binary relation $R$ on $\Lambda$ is called *compatible* (with the operations) if

$$\begin{aligned} M\, R\, N \;\Rightarrow\; & (ZM)\, R\, (ZN), \\ & (MZ)\, R\, (NZ) \text{ and} \\ & (\lambda x.M)\, R\, (\lambda x.N). \end{aligned}$$

(ii) A *congruence* relation on $\Lambda$ is a compatible equivalence relation.

(iii) A *reduction* relation on $\Lambda$ is a compatible, reflexive and transitive relation.

4.2. DEFINITION. The binary relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on $\Lambda$ are defined inductively as follows.

(i) 1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$;

2. $M \rightarrow_\beta N \;\Rightarrow\; ZM \rightarrow_\beta ZN$, $MZ \rightarrow_\beta NZ$ and $\lambda x.M \rightarrow_\beta \lambda x.N$.

(ii) 1. $M \twoheadrightarrow_\beta M$;

2. $M \rightarrow_\beta N \;\Rightarrow\; M \twoheadrightarrow_\beta N$;

3. $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \;\Rightarrow\; M \twoheadrightarrow_\beta L$.

(iii) 1. $M \twoheadrightarrow_\beta N \;\Rightarrow\; M =_\beta N$;
2. $M =_\beta N \;\Rightarrow\; N =_\beta M$;
3. $M =_\beta N, N =_\beta L \;\Rightarrow\; M =_\beta$L.

These relations are pronounced as follows.

$$\begin{array}{rcl} M \twoheadrightarrow_\beta N & : & M\beta\textit{-reduces} \text{ to } N; \\ M \rightarrow_\beta N & : & M\beta\textit{-reduces} \text{ to } N \textit{in one step}; \\ M =_\beta N & : & M \text{ is } \beta\textit{-convertible} \text{ to } N. \end{array}$$

By definition $\rightarrow_\beta$ is compatible, $\twoheadrightarrow_\beta$ is a reduction relation and $=_\beta$ is a congruence relation.

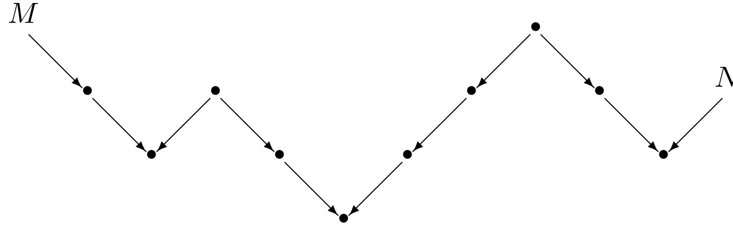
4.3. Example. (i) Define

$$\begin{array}{rcl} \boldsymbol{\omega} & \equiv & \lambda x.xx, \\ \boldsymbol{\Omega} & \equiv & \boldsymbol{\omega\omega}. \end{array}$$

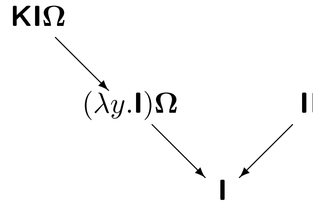
Then $\boldsymbol{\Omega} \rightarrow_\beta \boldsymbol{\Omega}$.

(ii) $\mathsf{K I}\boldsymbol{\Omega} \twoheadrightarrow_\beta \mathsf{I}$.

Intuitively, $M =_\beta N$ if $M$ is connected to $N$ via $\rightarrow_\beta$-arrows (disregarding the directions of these). In a picture this looks as follows.




4.4. Example. $\mathsf{K I}\boldsymbol{\Omega} =_\beta \mathsf{I I}$. This is demonstrated by the following reductions.




4.5. Proposition. $M =_\beta N \;\Leftrightarrow\; \boldsymbol{\lambda} \vdash M = N$.

Proof. By an easy induction. □

4.6. Definition. (i) A *β-redex* is a term of the form $(\lambda x.M)N$. In this case $M[x := N]$ is its *contractum.*

(ii) A λ-term $M$ is a *β-normal form* (β-nf) if it does not have a β-redex as subexpression.

(iii) A term $M$ *has* a β-normal form if $M =_\beta N$ and $N$ is a β-nf, for some $N$.

4.7. Example. $(\lambda x.xx)y$ is not a $\beta$-nf, but has as $\beta$-nf the term $yy$.

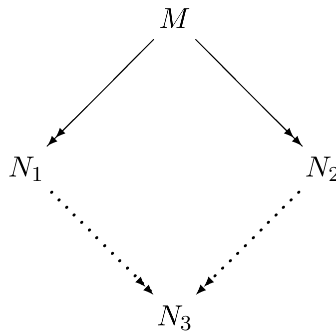
An immediate property of nf's is the following.

4.8. Lemma. *Let $M$ be a $\beta$-nf. Then*

$$M \twoheadrightarrow_\beta N \;\Rightarrow\; N \equiv M.$$

Proof. This is true if $\twoheadrightarrow_\beta$ is replaced by $\rightarrow_\beta$. Then the result follows by transitivity. □

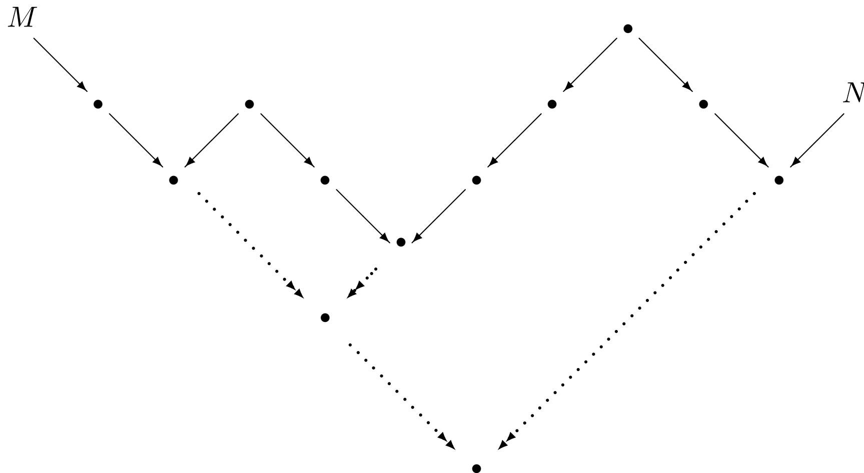
4.9. Church-Rosser Theorem. *If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some $N_3$ one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$; in diagram*




The proof is postponed until 4.19.

4.10. Corollary. *If $M =_\beta N$, then there is an $L$ such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$.*

An intuitive proof of this fact proceeds by a tiling procedure: given an arrow path showing $M =_\beta N$, apply the Church-Rosser property repeatedly in order to find a common reduct. For the example given above this looks as follows.




This is made precise below.