Introduction to Lambda Calculus

Henk Barendregt Erik Barendsen

Revised edition
December 1998, March 2000

Contents

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:

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

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:

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

or even by evaluating several expressions at the same time:

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

Application and abstraction

The first basic operation of the λ-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

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

where $[x := N]$ denotes substitution of N for x. It is remarkable that although (β) is the only essential axiom of the λ-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 (β). 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 λ-calculus will be introduced formally.

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

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

In BN-form this is

$$\begin{aligned} &\text{variable} \quad ::= \quad \text{`}v\text{'} \mid \text{variable `}'\text{'} \\ &\lambda\text{-term} \quad ::= \quad \text{variable} \mid \text{`(' } \lambda\text{-term } \lambda\text{-term `)'} \mid \text{`(}\lambda\text{' variable } \lambda\text{-term `)'} \end{aligned}$$

2.2. Example. The following are λ-terms.

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

2.3. Convention. (i) $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary λ-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.

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

(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 λ.

(ii) M is a *closed λ-term* (or *combinator*) if $\mathrm{FV}(M) = \emptyset$. The set of closed λ-terms is denoted by Λ^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 λ-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 λ-calculus as formal theory.

2.7. DEFINITION. (i) The principal axiom scheme of the λ-calculus is

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

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

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

Equality:

$$M = M;$$
$$M = N \quad \Rightarrow \quad N = M;$$
$$M = N, N = L \quad \Rightarrow \quad M = L.$$

Compatibility rules:

$$M = M' \quad \Rightarrow \quad MZ = M'Z;$$
$$M = M' \quad \Rightarrow \quad ZM = ZM';$$
$$M = M' \quad \Rightarrow \quad \lambda x.M = \lambda x.M'. \qquad (\xi)$$

(iii) If $M = N$ is provable in the λ-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 λ-calculus the following axiom scheme

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

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 λ-calculus the machine has to deal with this so called α-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 (β) 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

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

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

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

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 λ-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*

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

such that

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

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) $\exists G\, \forall X\ GX = \mathsf{S}GX$. Indeed,

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

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

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

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

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

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

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

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

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

(i) $\mathbf{A}_+ \boldsymbol{c}_n \boldsymbol{c}_m = \boldsymbol{c}_{n+m}$.
(ii) $\mathbf{A}_* \boldsymbol{c}_n \boldsymbol{c}_m = \boldsymbol{c}_{n*m}$.
(iii) $\mathbf{A}_{\exp} \boldsymbol{c}_n \boldsymbol{c}_m = \boldsymbol{c}_{(n^m)}$, *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

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

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

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

PROOF OF THE PROPOSITION. (i) Exercise.
(ii) Exercise. Use Lemma 2.16 (i).
(iii) By Lemma 2.16 (ii) we have for $m > 0$

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

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 $\mathbf{B} \equiv \lambda xyz.x(yz)$. Simplify $M \equiv \mathbf{B}XYZ$, 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) $M \equiv (\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\mathbf{I})$;
(iii) $M \equiv \mathbf{SKSKSK}$.

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

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

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

(ii) Construct a λ-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 K_∞;
(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 λ-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 λ-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 λ-terms exactly in the way they are written in Convention 2.3.

Chapter 3

The Power of Lambda

We have seen that the function plus, times and exponentiation on $\mathbb{N}$ can be represented in the λ-calculus using Church's numerals. We will now show that all computable (recursive) functions can be represented in the λ-calculus. In order to do this we will use first a different system of numerals.

Truth values and a conditional can be represented in the λ-calculus.

3.1. Definition. (i) $\mathbf{true} \equiv \mathbf{K}$, $\mathbf{false} \equiv \mathbf{K}_*$.

(ii) If B is considered as a Boolean, i.e. a term that is either **true** or **false**, then

$$\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

can be represented by

$$BPQ.$$

3.2. Definition (Pairing). For $M, N \in \Lambda$ write

$$[M, N] \equiv \lambda z.\mathsf{if}\ z\ \mathsf{then}\ M\ \mathsf{else}\ N \qquad (\equiv \lambda z.zMN).$$

Then

$$\begin{aligned} [M, N]\mathbf{true} &= M, \\ [M, N]\mathbf{false} &= N, \end{aligned}$$

and hence $[M, N]$ can serve as an ordered pair.

We can use this pairing construction for an alternative representation of natural numbers due to Barendregt (1976).

3.3. Definition. For each $n \in \mathbb{N}$, the numeral $\ulcorner n \urcorner$ is defined inductively as follows.

$$\begin{aligned} \ulcorner 0 \urcorner &\equiv \mathbf{I}, \\ \ulcorner n+1 \urcorner &\equiv [\mathbf{false}, \ulcorner n \urcorner]. \end{aligned}$$

3.4. Lemma (Successor, predecessor, test for zero). *There exist combinators* $\mathbf{S}^+$, $\mathbf{P}^-$, *and* **Zero** *such that*

$$\begin{aligned} \mathbf{S}^+\ulcorner n\urcorner &= \ulcorner n+1\urcorner, \\ \mathbf{P}^-\ulcorner n+1\urcorner &= \ulcorner n\urcorner, \\ \mathbf{Zero}\ulcorner 0\urcorner &= \mathbf{true}, \\ \mathbf{Zero}\ulcorner n+1\urcorner &= \mathbf{false}. \end{aligned}$$

Proof. Take

$$\begin{aligned} \mathbf{S}^+ &\equiv \lambda x.[\mathbf{false}, x], \\ \mathbf{P}^- &\equiv \lambda x.x\,\mathbf{false}, \\ \mathbf{Zero} &\equiv \lambda x.x\,\mathbf{true}. \ \square \end{aligned}$$

3.5. Definition (Lambda definability). (i) A *numeric function* is a map

$$\varphi : \mathbb{N}^p \to \mathbb{N}$$

for some p. In this case φ is called p-ary.

(ii) A numeric p-ary function φ is called λ-*definable* if for some combinator F

$$F\ulcorner n_1\urcorner \ldots \ulcorner n_p\urcorner = \ulcorner \varphi(n_1,\ldots,n_p)\urcorner \qquad (*)$$

for all $n_1,\ldots,n_p \in \mathbb{N}$. If $(*)$ holds, then φ is said to be λ-*defined* by F.

3.6. Definition. The *initial functions* are the numeric functions U_i^n, S^+, Z defined by

$$\begin{aligned} U_i^n(x_1,\ldots,x_n) &= x_i, \quad (1 \le i \le n); \\ S^+(n) &= n+1; \\ Z(n) &= 0. \end{aligned}$$

Let P(n) be a numeric relation. As usual

$$\mu m[P(m)]$$

denotes the least number m such that $P(m)$ holds if there is such a number; otherwise it is undefined.

3.7. Definition. Let $\mathcal{A}$ be a class of numeric functions.

(i) $\mathcal{A}$ is *closed under composition* if for all φ defined by

$$\varphi(\vec{n}) = \chi(\psi_1(\vec{n}),\ldots,\psi_m(\vec{n}))$$

with $\chi,\psi_1,\ldots,\psi_m \in \mathcal{A}$, one has $\varphi \in \mathcal{A}$.

(ii) $\mathcal{A}$ is *closed under primitive recursion* if for all φ defined by

$$\begin{aligned} \varphi(0,\vec{n}) &= \chi(\vec{n}), \\ \varphi(k+1,\vec{n}) &= \psi(\varphi(k,\vec{n}),k,\vec{n}) \end{aligned}$$

with $\chi,\psi \in \mathcal{A}$, one has $\varphi \in \mathcal{A}$.

(iii) $\mathcal{A}$ is *closed under minimalization* if for all φ defined by

$$\varphi(\vec{n}) = \mu m[\chi(\vec{n}, m) = 0]$$

with $\chi \in \mathcal{A}$ such that

$$\forall \vec{n}\ \exists m\ \chi(\vec{n}, m) = 0,$$

one has $\varphi \in \mathcal{A}$.

3.8. Definition. The class $\mathcal{R}$ of *recursive functions* is the smallest class of numeric functions that contains all initial functions and is closed under composition, primitive recursion and minimalization.

So $\mathcal{R}$ is an inductively defined class. The proof that all recursive functions are λ-definable is in fact by a corresponding induction argument. The result is originally due to Kleene (1936).

3.9. Lemma. *The initial functions are λ-definable.*

Proof. Take as defining terms

$$\begin{array}{rcl} \mathbf{U}_i^n & \equiv & \lambda x_1 \cdots x_n.x_i, \\ \mathbf{S}^+ & \equiv & \lambda x.[\mathbf{false}, x] \quad \text{(see Lemma 3.4)} \\ \mathbf{Z} & \equiv & \lambda x.\ulcorner 0 \urcorner. \ \square \end{array}$$

3.10. Lemma. *The λ-definable functions are closed under composition.*

Proof. Let $\chi, \psi_1, \ldots, \psi_m$ be λ-defined by $G, H_1, \ldots, H_m$ respectively. Then

$$\varphi(\vec{n}) = \chi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

is λ-defined by

$$F \equiv \lambda \vec{x}.G(H_1\vec{x}) \cdots (H_m\vec{x}). \ \square$$

As to primitive recursion, let us first consider an example. The addition function can be specified as follows.

$$\begin{array}{rcl} \mathrm{Add}(0, y) & = & y, \\ \mathrm{Add}(x+1, y) & = & 1 + \mathrm{Add}(x, y) \ = \ S^+(\mathrm{Add}(x, y)). \end{array}$$

An intuitive way to compute $\mathrm{Add}(m, n)$ us the following.

Test whether $m = 0$.
If yes: give output n;
if no: compute $\mathrm{Add}(m-1, n)$ and give its successor as output.

Therefore we want a term **Add** such that

$$\mathbf{Add}\, xy = \mathsf{if}\ \mathbf{Zero}\, x\ \mathsf{then}\ y\ \mathsf{else}\ \mathbf{S}^+(\mathbf{Add}(\mathbf{P}^- x)y).$$

This equation can be solved using the fixedpoint combinator: take

$$\mathbf{Add} \equiv \mathbf{Y}(\lambda axy.\mathsf{if}\ \mathbf{Zero}\, x\ \mathsf{then}\ y\ \mathsf{else}\ \mathbf{S}^+(a(\mathbf{P}^- x)y)).$$

The general case is treated as follows.

3.11. Lemma. *The λ-definable functions are closed under primitive recursion.*

Proof. Let φ be defined by

$$\begin{aligned} \varphi(0,\vec{n}) &= \chi(\vec{n}), \\ \varphi(k+1,\vec{n}) &= \psi(\varphi(k,\vec{n}),k,\vec{n}), \end{aligned}$$

where χ,ψ are λ-defined by G,H respectively. Now we want a term F such that

$$\begin{aligned} Fx\vec{y} &= \mathsf{if}\ \mathbf{Zero}\,x\ \mathsf{then}\ G\vec{y}\ \mathsf{else}\ H(F(\mathbf{P}^-x)\vec{y})(\mathbf{P}^-x)\vec{y} \\ &\equiv D(F,x,\vec{y}), \text{ say.} \end{aligned}$$

It is sufficient to find an F such that

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

Now such an F can be found by the Fixedpoint Theorem and we are done. □

3.12. Lemma. *The λ-definable functions are closed under minimalization.*

Proof. Let φ be defined by

$$\varphi(\vec{n}) = \mu m[\chi(\vec{n},m)=0],$$

where χ is λ-defined by G. Again by the Fixedpoint Theorem there is a term H such that

$$\begin{aligned} H\vec{x}y &= \mathsf{if}\ \mathbf{Zero}(G\vec{x}y)\ \mathsf{then}\ y\ \mathsf{else}\ H\vec{x}(\mathbf{S}^+y) \\ &= (\lambda h\vec{x}y.E(h,\vec{x},y))H\vec{x}y, \text{ say.} \end{aligned}$$

Set $F \equiv \lambda\vec{x}.H\vec{x}\ulcorner 0\urcorner$. Then F λ-defines φ:

$$\begin{aligned} F\ulcorner\vec{n}\urcorner &= H\ulcorner\vec{n}\urcorner\ulcorner 0\urcorner \\ &= \ulcorner 0\urcorner && \text{if } G\ulcorner\vec{n}\urcorner\ulcorner 0\urcorner = \ulcorner 0\urcorner \\ &= H\ulcorner\vec{n}\urcorner\ulcorner 1\urcorner && \text{else} \\ &= \ulcorner 1\urcorner && \text{if } G\ulcorner\vec{n}\urcorner\ulcorner 1\urcorner = \ulcorner 0\urcorner \\ &= H\ulcorner\vec{n}\urcorner\ulcorner 2\urcorner && \text{else} \\ &= \ulcorner 2\urcorner && \text{if } \ldots \\ &= \ldots \quad \square \end{aligned}$$

3.13. Theorem. *All recursive functions are λ-definable.*

Proof. By the lemmas 3.9–3.12. □

The converse also holds. So for numeric functions we have φ is recursive iff φ is λ-definable. Moreover also for partial functions a notion of λ-definability exists. If ψ is a partial numeric function, then we have

$$\psi \text{ is partial recursive } \Leftrightarrow \ \psi \text{ is } \lambda\text{-definable.}$$

3.14. Theorem. *With respect to the Church numerals $\boldsymbol{c}_n$ all recursive functions can be λ-defined.*

Proof. Define

$$\begin{aligned}\mathsf{S}^+_{\boldsymbol{c}} &\equiv \lambda xyz.y(xyz),\\ \mathsf{P}^-_{\boldsymbol{c}} &\equiv \lambda xyz.x(\lambda pq.q(py))(\mathsf{K}z)\mathsf{I}^{1},\\ \mathsf{Zero}_{\boldsymbol{c}} &\equiv \lambda x.x(\mathsf{K}\,\mathsf{false})\mathsf{true}.\end{aligned}$$

Then these terms represent the successor, predecessor and test for zero. Then as before all recursive functions can be λ-defined. □

An alternative proof uses 'translators' between the numerals $\ulcorner n \urcorner$ and $\boldsymbol{c}_n$.

3.15. Proposition. *There exist terms T, T^{-1} such that for all n*

$$\begin{aligned}T\boldsymbol{c}_n &= \ulcorner n \urcorner;\\ T^{-1}\ulcorner n \urcorner &= \boldsymbol{c}_n.\end{aligned}$$

Proof. Construct T, T^{-1} such that

$$\begin{aligned}T &\equiv \lambda x.x\mathsf{S}^+\ulcorner 0\urcorner.\\ T^{-1} &= \lambda x.\mathsf{if}\ \mathsf{Zero}\ x\ \mathsf{then}\ \boldsymbol{c}_0\ \mathsf{else}\ \mathsf{S}^+_{\boldsymbol{c}}(T^{-1}(\mathsf{P}^-x)).\ \square\end{aligned}$$

3.16. Corollary (Second proof of Theorem 3.14). *Let φ be a recursive function (of arity 2 say). Let F represent φ with respect to the numerals $\ulcorner n \urcorner$. Define*

$$F_{\boldsymbol{c}} \equiv \lambda xy.T^{-1}(F(Tx)(Ty)).$$

Then $F_{\boldsymbol{c}}$ represents φ with respect to the Church numerals. □

The representation of pairs in the lambda calculus can also be used to solve multiple fixedpoint equations.

3.17. Multiple Fixedpoint Theorem. *Let $F_1, \ldots, F_n$ be λ-terms. Then we can find $X_1, \ldots, X_n$ such that*

$$\begin{aligned}X_1 &= F_1X_1\cdots X_n,\\ &\vdots\\ X_n &= F_nX_1\cdots X_n.\end{aligned}$$

Observe that for $n = 1$ this is the ordinary Fixedpoint Theorem (2.12).

Proof. We treat the case $n = 2$. So we want

$$\begin{aligned}X_1 &= F_1X_1X_2,\\ X_2 &= F_2X_1X_2.\end{aligned}$$

[1] Term found by J. Velmans.

The trick is to construct X_1 and X_2 simultaneously, as a pair. By the ordinary Fixedpoint Theorem we can find an X such that

$$X = [F_1(X\mathbf{true})(X\mathbf{false}), F_2(X\mathbf{true})(X\mathbf{false})].$$

Now define $X_1 \equiv X\mathbf{true}$, $X_2 \equiv X\mathbf{false}$. Then the result follows. This can be generalized to arbitrary n. □

3.18. Example. There exist $G, H \in \Lambda$ such that

$$\begin{aligned} Gxy &= Hy(\mathbf{K}x), \\ Hx &= G(xx)(\mathbf{S}(H(xx))). \end{aligned}$$

Indeed, we can replace the above equations by

$$\begin{aligned} G &= \lambda xy.Hy(\mathbf{K}x), \\ H &= \lambda x.G(xx)(\mathbf{S}(H(xx))), \end{aligned}$$

and apply the Multiple Fixedpoint Theorem with $F_1 \equiv \lambda ghxy.hy(\mathbf{K}x)$ and $F_2 \equiv \lambda ghx.g(xx)(\mathbf{S}(h(xx)))$.

Exercises

3.1. (i) Find a λ-term **Mult** such that for all $m, n \in \mathbb{N}$

$$\mathbf{Mult}\ulcorner n\urcorner\ulcorner m\urcorner = \ulcorner n \cdot m\urcorner.$$

(ii) Find a λ-term **Fac** such that for all $n \in \mathbb{N}$

$$\mathbf{Fac}\ulcorner n\urcorner = \ulcorner n!\urcorner.$$

3.2. The *simple Ackermann function* φ is defined as follows.

$$\begin{aligned} \varphi(0, n) &= n + 1, \\ \varphi(m + 1, 0) &= \varphi(m, 1), \\ \varphi(m + 1, n + 1) &= \varphi(m, \varphi(m + 1, n)). \end{aligned}$$

Find a λ-term F that λ-defines φ.

3.3. Construct λ-terms $M_0, M_1, \ldots$ such that for all n one has

$$\begin{aligned} M_0 &= x, \\ M_{n+1} &= M_{n+2}M_n. \end{aligned}$$

3.4. Verify that $\mathbf{P}^-_{\mathbf{c}}$ (see the first proof of Theorem 3.14) indeed λ-defines the predecessor function with respect to the Church numerals.

Chapter 4

Reduction

There is a certain asymmetry in the basic scheme (β). 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 Λ is called *compatible* (with the operations) if

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

(ii) A *congruence* relation on Λ is a compatible equivalence relation.

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

4.2. Definition. The binary relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ 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.

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

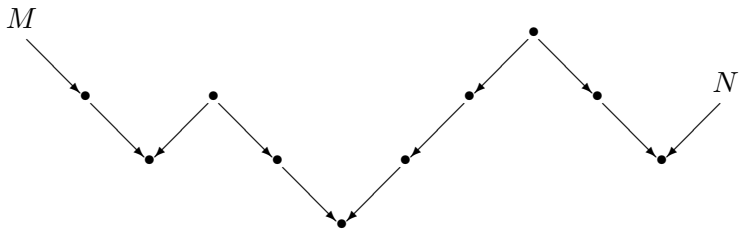
By definition $\to_\beta$ is compatible, $\twoheadrightarrow_\beta$ is a reduction relation and $=_\beta$ is a congruence relation.

4.3. Example. (i) Define

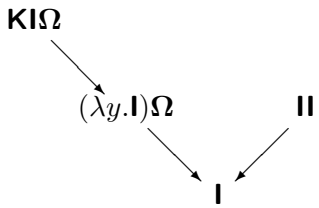
$$\begin{array}{rcl} \boldsymbol{\omega} & \equiv & \lambda x.xx, \\ \boldsymbol{\Omega} & \equiv & \boldsymbol{\omega\omega}. \end{array}$$

Then $\boldsymbol{\Omega} \to_\beta \boldsymbol{\Omega}$.
(ii) $\mathbf{KI\Omega} \twoheadrightarrow_\beta \mathbf{I}$.

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



4.4. Example. $\mathbf{KI\Omega} =_\beta \mathbf{II}$. 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 β-nf, but has as β-nf the term yy.

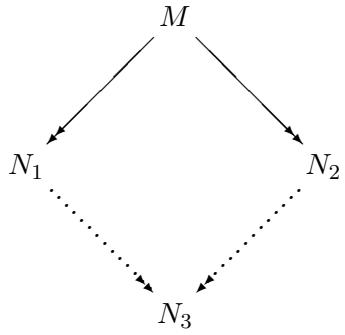
An immediate property of nf's is the following.

4.8. LEMMA. *Let M be a β-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. □

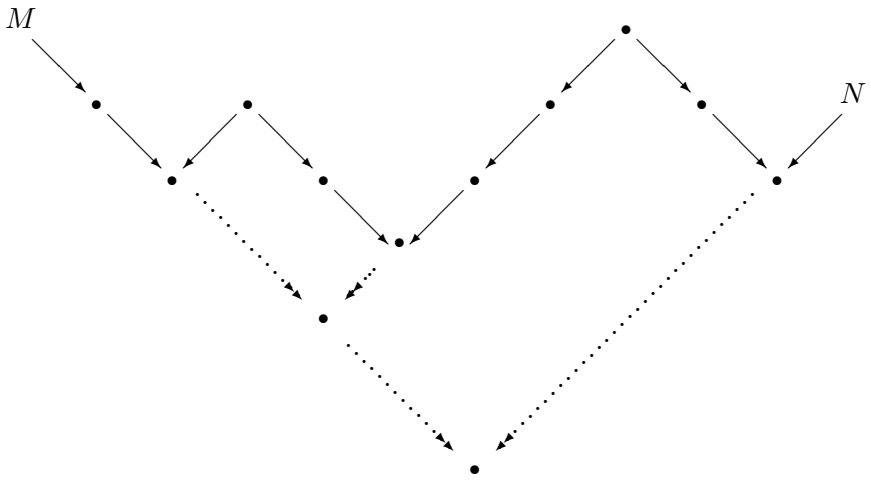
4.9. CHURCH-ROSSER THEOREM. *If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$; in diagram*



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.



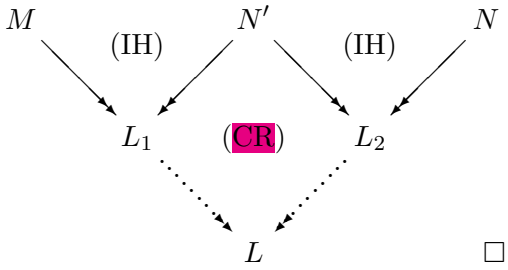
This is made precise below.

PROOF. Induction on the generation of $=_\beta$.

Case 1. $M =_\beta N$ because $M \twoheadrightarrow_\beta N$. Take $L \equiv N$.

Case 2. $M =_\beta N$ because $N =_\beta M$. By the IH there is a common β-reduct L_1 of N, M. Take $L \equiv L_1$.

Case 3. $M =_\beta N$ because $M =_\beta N'$, $N' =_\beta N$. Then



□

4.11. COROLLARY. (i) *If M has N as β-nf, then $M \twoheadrightarrow_\beta N$.*

(ii) *A λ-term has at most one β-nf.*

PROOF. (i) Suppose $M =_\beta N$ with N in β-nf. By Corollary 4.10 $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$ for some L. But then $N \equiv L$, by Lemma 4.8, so $M \twoheadrightarrow_\beta N$.

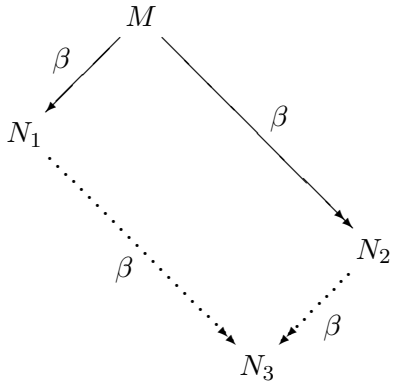
(ii) Suppose M has β-nf's N_1, N_2. Then $N_1 =_\beta N_2$ $(=_\beta M)$. By Corollary 4.10 $N_1 \twoheadrightarrow_\beta L$, $N_2 \twoheadrightarrow_\beta L$ for some L. But then $N_1 \equiv L \equiv N_2$ by Lemma 4.8. □

4.12. SOME CONSEQUENCES. (i) The λ-calculus is consistent, i.e. $\boldsymbol{\lambda} \nvdash \mathbf{true} = \mathbf{false}$. Otherwise $\mathbf{true} =_\beta \mathbf{false}$ by Proposition 4.5, which is impossible by Corollary 4.11 since **true** and **false** are distinct β-nf's. This is a syntactic consistency proof.

(ii) $\boldsymbol{\Omega}$ has no β-nf. Otherwise $\boldsymbol{\Omega} \twoheadrightarrow_\beta N$ with N in β-nf. But $\boldsymbol{\Omega}$ only reduces to itself and is not in β-nf.

(iii) In order to find the β-nf of a term M (if it exists), the various subexpressions of M may be reduced in different orders. By Corollary 4.11 (ii) the β-nf is unique.

The proof of the Church-Rosser theorem occupies 4.13–4.19. The idea of the proof is as follows. In order to prove Theorem 4.9, it is sufficient to show the Strip Lemma:



In order to prove this lemma, let $M \to_\beta N_1$ be a one step reduction resulting from changing a redex R in M in its contractum R' in N_1. If one makes a

bookkeeping of what happens with R during the reduction $M \twoheadrightarrow_\beta N_2$, then by reducing all 'residuals' of R in N_2 the term N_3 can be found. In order to do the necessary bookkeeping an extended set $\underline{\Lambda} \supseteq \Lambda$ and reduction $\underline{\beta}$ is introduced. The underlining serves as a 'tracing isotope'.

4.13. Definition (Underlining). (i) $\underline{\Lambda}$ is the set of terms defined inductively as follows.

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

(ii) The underlined reduction relations $\to_{\underline{\beta}}$ (one step) and $\twoheadrightarrow_{\underline{\beta}}$ are defined starting with the contraction rules

$$\begin{array}{rcl} (\lambda x.M)N & \to_{\underline{\beta}} & M[x := N], \\ (\underline{\lambda} x.M)N & \to_{\underline{\beta}} & M[x := N]. \end{array}$$

Then $\to_{\underline{\beta}}$ is extended in order to become a compatible relation (also with respect to $\underline{\lambda}$-abstraction). Moreover, $\twoheadrightarrow_{\underline{\beta}}$ is the transitive reflexive closure of $\to_{\underline{\beta}}$.

(iii) If $M \in \underline{\Lambda}$, then $|M| \in \Lambda$ is obtained from M by leaving out all underlinings. E.g. $|(\lambda x.x)((\underline{\lambda} x.x)(\lambda x.x))| \equiv \mathbf{I}(\mathbf{II})$.

4.14. Definition. The map $\varphi : \underline{\Lambda} \to \Lambda$ is defined inductively as follows.

$$\begin{array}{rcl} \varphi(x) & \equiv & x, \\ \varphi(MN) & \equiv & \varphi(M)\varphi(N), \\ \varphi(\lambda x.M) & \equiv & \lambda x.\varphi(M), \\ \varphi((\underline{\lambda} x.M)N) & \equiv & \varphi(M)[x := \varphi(N)]. \end{array}$$

In other words, φ contracts all redexes that are underlined, from the inside to the outside.

Notation. If $|M| \equiv N$ or $\varphi(M) \equiv N$, then this will be denoted by

$$M \xrightarrow[|\,|]{} N \text{ or } M \xrightarrow[\varphi]{} N.$$

4.15. Lemma.

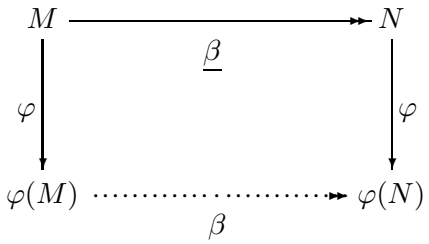
$$\begin{array}{ccc} M' & \overset{\underline{\beta}}{\cdots\cdots\twoheadrightarrow} & N' \\ {\scriptstyle |\,|}\downarrow & & \downarrow{\scriptstyle |\,|} \\ M & \underset{\beta}{\twoheadrightarrow} & N \end{array} \qquad M', N' \in \underline{\Lambda}, \; M, N \in \Lambda.$$

PROOF. First suppose $M \to_\beta N$. Then N is obtained by contracting a redex in M and N' can be obtained by contracting the corresponding redex in M'. The general statement follows by transitivity. □

4.16. LEMMA. (i) *Let* $M, N \in \underline{\Lambda}$. *Then*

$$\varphi(M[x := N]) \equiv \varphi(M)[x := \varphi(N)].$$

(ii)

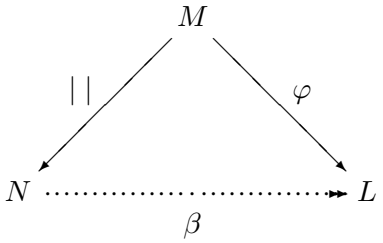


$M, N \in \underline{\Lambda}$.

PROOF. (i) By induction on the structure of M, using the Substitution Lemma (see Exercise 2.2) in case $M \equiv (\underline{\lambda} y.P)Q$. The condition of that lemma may be assumed to hold by our convention about free variables.

(ii) By induction on the generation of $\twoheadrightarrow_{\underline{\beta}}$, using (i). □

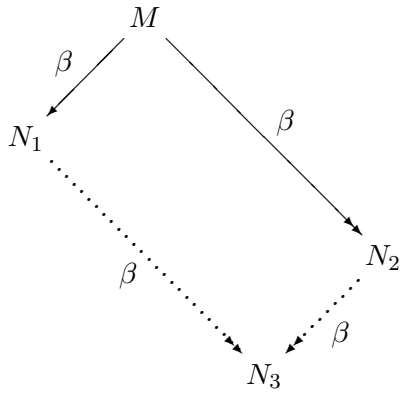
4.17. LEMMA.



$M \in \underline{\Lambda}$,
$N, L \in \Lambda$.

PROOF. By induction on the structure of M. □

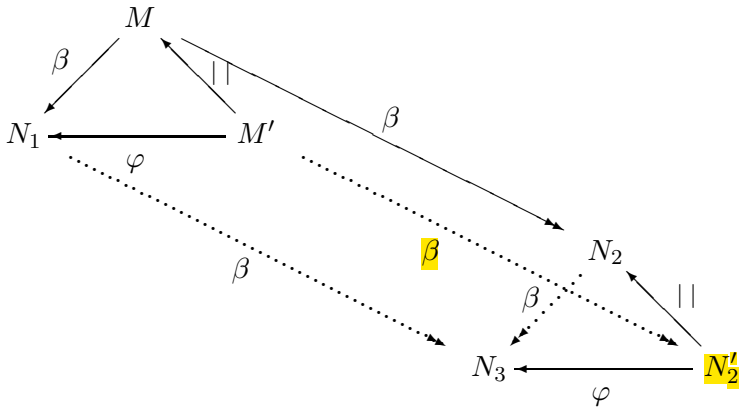
4.18. STRIP LEMMA.



$M, N_1, N_2, N_3 \in \Lambda$.

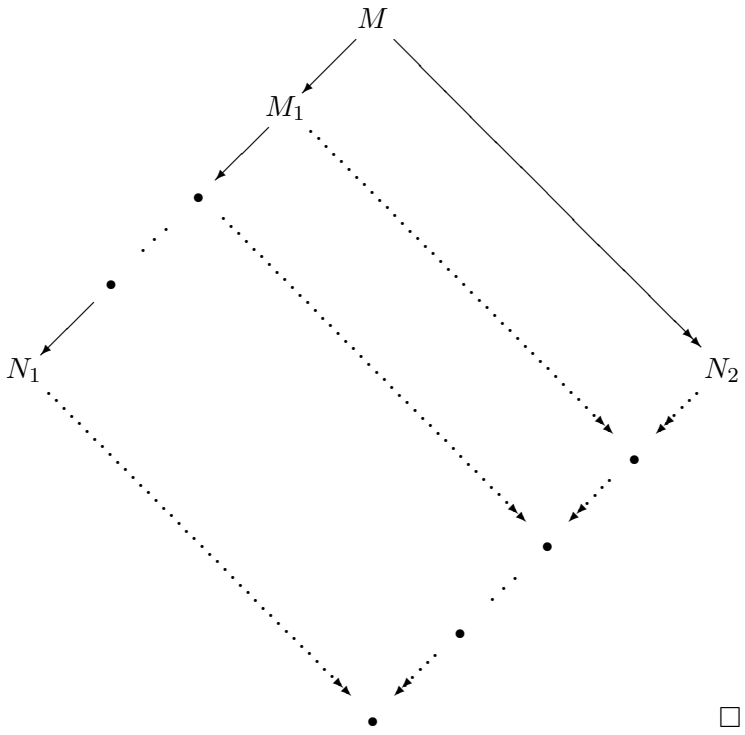
PROOF. Let N_1 be the result of contracting the redex occurrence $R \equiv (\lambda x.P)Q$ in M. Let $M' \in \underline{\Lambda}$ be obtained from M by replacing R by $R' \equiv (\underline{\lambda} x.P)Q$. Then

$|M'| \equiv M$ and $\varphi(M') \equiv N_1$. By the lemmas 4.15, 4.16 and 4.17 we can erect the diagram



which proves the Strip Lemma. □

4.19. Proof of the Church-Rosser Theorem. If $M \twoheadrightarrow_\beta N_1$, then $M \equiv M_1 \to_\beta M_2 \to_\beta \cdots \to_\beta M_n \equiv N_1$. Hence the CR property follows from the Strip Lemma and a simple diagram chase:



□

4.20. Definition. For $M \in \Lambda$ the *reduction graph* of M, notation $G_\beta(M)$, is the directed multigraph with vertices $\{N \mid M \twoheadrightarrow_\beta N\}$ and directed by $\to_\beta$.

4.21. EXAMPLE. $G_\beta(\mathsf{I}(\mathsf{I}x))$ is

sometimes simply drawn as

It can happen that a term M has a nf, but at the same time an infinite reduction path. Let $\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$. Then $\mathbf{\Omega} \to \mathbf{\Omega} \to \cdots$ so $\mathbf{KI\Omega} \to \mathbf{KI\Omega} \to \cdots$, and $\mathbf{KI\Omega} \twoheadrightarrow \mathbf{I}$. Therefore a so called *strategy* is necessary in order to find the normal form. We state the following theorem; for a proof see Barendregt (1984), Theorem 13.2.2.

4.22. NORMALIZATION THEOREM. *If M has a normal form, then iterated contraction of the leftmost redex leads to that normal form.*

In other words: the leftmost reduction strategy is *normalizing*. This fact can be used to find the normal form of a term, or to prove that a certain term has no normal form.

4.23. EXAMPLE. $\mathbf{K\Omega I}$ has an infinite leftmost reduction path, viz.

$$\mathbf{K\Omega I} \to_\beta (\lambda y.\mathbf{\Omega})\mathbf{I} \to_\beta \mathbf{\Omega} \to_\beta \mathbf{\Omega} \to_\beta \cdots,$$

and hence does not have a normal form.

The functional language (pure) *Lisp* uses an *eager* or *applicative* evaluation strategy, i.e. whenever an expression of the form FA has to be evaluated, A is reduced to normal form first, before 'calling' F. In the λ-calculus this strategy is not normalizing as is shown by the two reduction paths for $\mathbf{KI\Omega}$ above. There is, however, a variant of the lambda calculus, called the λI-calculus, in which the eager evaluation strategy is normalizing. In this λI-calculus terms like $\mathbf{K}$, 'throwing away' $\mathbf{\Omega}$ in the reduction $\mathbf{KI\Omega} \twoheadrightarrow \mathbf{I}$ do not exist. The 'ordinary' λ-calculus is sometimes referred to as λK-calculus; see Barendregt (1984), Chapter 9.

Remember the fixedpoint combinator $\mathbf{Y}$. For each $F \in \Lambda$ one has $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$, but neither $\mathbf{Y}F \twoheadrightarrow_\beta F(\mathbf{Y}F)$ nor $F(\mathbf{Y}F) \twoheadrightarrow_\beta \mathbf{Y}F$. In order to solve

reduction equations one can work with A.M. Turing's fixedpoint combinator, which has a different reduction behaviour.

4.24. Definition. Turing's fixedpoint combinator $\mathbf{\Theta}$ is defined by setting

$$\begin{aligned} A &\equiv \lambda xy.y(xxy), \\ \mathbf{\Theta} &\equiv AA. \end{aligned}$$

4.25. Proposition. *For all $F \in \Lambda$ one has*

$$\mathbf{\Theta}F \twoheadrightarrow_\beta F(\mathbf{\Theta}F).$$

Proof.

$$\begin{aligned} \mathbf{\Theta}F &\equiv AAF \\ &\to_\beta (\lambda y.y(AAy))F \\ &\to_\beta F(AAF) \\ &\equiv F(\mathbf{\Theta}F). \ \square \end{aligned}$$

4.26. Example. $\exists G\, \forall X\, GX \twoheadrightarrow X(XG)$. Indeed,

$$\begin{aligned} \forall X\, GX \twoheadrightarrow X(XG) &\Leftarrow G \twoheadrightarrow \lambda x.x(xG) \\ &\Leftarrow G \twoheadrightarrow (\lambda gx.x(xg))G \\ &\Leftarrow G \equiv \mathbf{\Theta}(\lambda gx.x(xg)). \end{aligned}$$

Also the Multiple Fixedpoint Theorem has a 'reducing' variant.

4.27. Theorem. *Let $F_1, \ldots, F_n$ be λ-terms. Then we can find $X_1, \ldots, X_n$ such that*

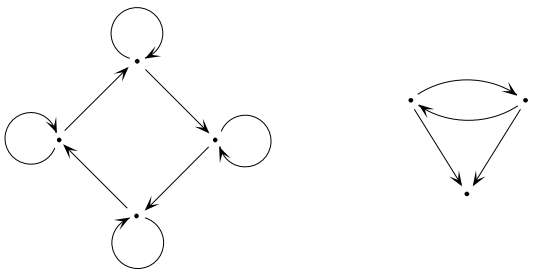
$$\begin{aligned} X_1 &\twoheadrightarrow F_1X_1 \cdots X_n, \\ &\vdots \\ X_n &\twoheadrightarrow F_nX_1 \cdots X_n. \end{aligned}$$

Proof. As for the equational Multiple Fixedpoint Theorem 3.17, but now using $\mathbf{\Theta}$. $\square$

Exercises

4.1. Show $\forall M\, \exists N$ [N in β-nf and $N\mathbf{I} \twoheadrightarrow_\beta M$].

4.2. Construct four terms M with $G_\beta(M)$ respectively as follows.



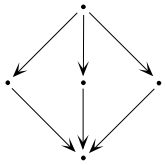
4.3. Show that there is no $F \in \Lambda$ such that for all $M, N \in \Lambda$

$$F(MN) = M.$$

4.4.* Let $M \equiv AAx$ with $A \equiv \lambda axz.z(aax)$. Show that $G_\beta(M)$ contains as subgraphs an n-dimensional cube for every $n \in \mathbb{N}$.

4.5. (A. Visser)
(i) Show that there is only one redex R such that $G_\beta(R)$ is as follows.

(ii) Show that there is no $M \in \Lambda$ with $G_\beta(M)$ is



[*Hint.* Consider the relative positions of redexes.]

4.6.* (C. Böhm) Examine $G_\beta(M)$ with M equal to
(i) $H\mathbf{I}H, \quad H \equiv \lambda xy.x(\lambda z.yzy)x.$
(ii) $LL\mathbf{I}, \quad L \equiv \lambda xy.x(yy)x.$
(iii) $Q\mathbf{I}Q, \quad Q \equiv \lambda xy.xy\mathbf{I}xy.$

4.7.* (J.W. Klop) Extend the λ-calculus with two constants $\boldsymbol{\delta}$, $\boldsymbol{\varepsilon}$. The reduction rules are extended to include $\boldsymbol{\delta}MM \to \boldsymbol{\varepsilon}$. Show that the resulting system is not Church-Rosser.
[*Hint.* Define terms C, D such that

$$\begin{aligned} Cx &\twoheadrightarrow \boldsymbol{\delta}x(Cx) \\ D &\twoheadrightarrow CD \end{aligned}$$

Then $D \twoheadrightarrow \boldsymbol{\varepsilon}$ and $D \twoheadrightarrow C\boldsymbol{\varepsilon}$ in the extended reduction system, but there is no common reduct.]

4.8. Show that the term $M \equiv AAx$ with $A \equiv \lambda axz.z(aax)$ does not have a normal form.

4.9. (i) Show $\boldsymbol{\lambda} \nvdash WWW = \boldsymbol{\omega}_3\boldsymbol{\omega}_3$, with $W \equiv \lambda xy.xyy$ and $\boldsymbol{\omega}_3 \equiv \lambda x.xxx$.
(ii) Show $\boldsymbol{\lambda} \nvdash B_x = B_y$ with $B_z \equiv A_zA_z$ and $A_z \equiv \lambda p.ppz$.

4.10. Draw $G_\beta(M)$ for M equal to:
(i) $WWW, \quad W \equiv \lambda xy.xyy.$
(ii) $\boldsymbol{\omega}\boldsymbol{\omega}, \quad \boldsymbol{\omega} \equiv \lambda x.xx.$
(iii) $\boldsymbol{\omega}_3\boldsymbol{\omega}_3, \quad \boldsymbol{\omega}_3 \equiv \lambda x.xxx.$
(iv) $(\lambda x.\mathbf{I}xx)(\lambda x.\mathbf{I}xx)$.
(v) $(\lambda x.\mathbf{I}(xx))(\lambda x.\mathbf{I}(xx))$.
(vi) $\mathbf{II}(\mathbf{II})$.

4.11. The *length* of a term is its number of symbols times 0.5 cm. Write down a λ-term of length < 30 cm with normal form $> 10^{10^{10}}$ light year.
[*Hint.* Use Proposition 2.15 (ii). The speed of light is $c = 3 \times 10^{10}$ cm/s.]

Chapter 5

Type Assignment

The lambda calculus as treated so far is usually referred to as a *type-free* theory. This is so, because every expression (considered as a function) may be applied to every other expression (considered as an argument). For example, the identity function $\mathsf{I} \equiv \lambda x.x$ may be applied to any argument x to give as result that same x. In particular I may be applied to itself.

There are also typed versions of the lambda calculus. These are introduced essentially in Curry (1934) (for the so called Combinatory Logic, a variant of the lambda calculus) and in Church (1940). Types are usually objects of a syntactic nature and may be assigned to lambda terms. If M is such a term and a type A is assigned to M, then we say 'M *has type* A' or 'M *in* A'; the denotation used for this is $M : A$. For example in some typed systems one has $\mathsf{I} : (A \to A)$, that is, the identity I may get as type $A \to A$. This means that if x being an argument of I is of type A, then also the value $\mathsf{I}x$ is of type A. In general, $A \to B$ is the type of functions from A to B.

Although the analogy is not perfect, the type assigned to a term may be compared to the dimension of a physical entity. These dimensions prevent us from wrong operations like adding 3 volt to 2 ampère. In a similar way types assigned to lambda terms provide a partial specification of the algorithms that are represented and are useful for showing partial correctness.

Types may also be used to improve the efficiency of compilation of terms representing functional algorithms. If for example it is known (by looking at types) that a subexpression of a term (representing a funtional program) is purely arithmetical, then fast evaluation is possible. This is because the expression then can be executed by the ALU of the machine and not in the slower way in which symbolic expressions are evaluated in general.

The two original papers of Curry and Church introducing typed versions of the lambda calculus give rise to two different families of systems. In the typed lambda calculi *à la* Curry terms are those of the type-free theory. Each term has a set of possible types. This set may be empty, be a singleton or consist of several (possibly infinitely many) elements. In the systems *à la* Church the terms are annotated versions of the type-free terms. Each term has (up to an equivalence relation) a unique type that is usually derivable from the way the term is annotated.

The Curry and Church approaches to typed lambda calculus correspond to

two paradigms in programming. In the first of these a program may be written without typing at all. Then a compiler should check whether a type can be assigned to the program. This will be the case if the program is correct. A well-known example of such a language is *ML*, see Milner (1984). The style of typing is called *implicit typing*. The other paradigm in programming is called *explicit typing* and corresponds to the Church version of typed lambda calculi. Here a program should be written together with its type. For these languages type-checking is usually easier, since no types have to be constructed. Examples of such languages are *Algol 68* and *Pascal*. Some authors designate the Curry systems as 'lambda calculi *with type assignment*' and the Church systems as 'systems of *typed* lambda calculus'.

Within each of the two paradigms there are several versions of typed lambda calculus. In many important systems, especially those *à la* Church, it is the case that terms that do have a type always possess a normal form. By the unsolvability of the halting problem this implies that not all computable functions can be represented by a typed term, see Barendregt (1990), Theorem 4.2.15. This is not so bad as it sounds, because in order to find such computable functions that cannot be represented, one has to stand on one's head. For example in $\lambda 2$, the second order typed lambda calculus, only those partial recursive functions cannot be represented that happen to be total, but not provably so in mathematical analysis (second order arithmetic).

Considering terms and types as programs and their specifications is not the only possibility. A type A can also be viewed as a proposition and a term M in A as a proof of this proposition. This so called propositions-as-types interpretation is independently due to de Bruijn (1970) and Howard (1980) (both papers were conceived in 1968). Hints in this direction were given in Curry and Feys (1958) and in Läuchli (1970). Several systems of proof checking are based on this interpretation of propositions-as-types and of proofs-as-terms. See e.g. de Bruijn (1980) for a survey of the so called AUTOMATH proof checking system. Normalization of terms corresponds in the formulas-as-types interpretation to normalisation of proofs in the sense of Prawitz (1965). Normal proofs often give useful proof theoretic information, see e.g. Schwichtenberg (1977).

In this section a typed lambda calculus will be introduced in the style of Curry. For more information, see Barendregt (1992).

The system $\lambda\!\!\rightarrow$-Curry

Originally the implicit typing paradigm was introduced in Curry (1934) for the theory of combinators. In Curry and Feys (1958) and Curry et al. (1972) the theory was modified in a natural way to the lambda calculus assigning elements of a given set $\mathbb{T}$ of types to type free lambda terms. For this reason these calculi *à la* Curry are sometimes called *systems of type assignment*. If the type $\sigma \in \mathbb{T}$ is assigned to the term $M \in \Lambda$ one writes $\vdash M : \sigma$, sometimes with a subscript under $\vdash$ to denote the particular system. Usually a set of assumptions Γ is needed to derive a type assignment and one writes $\Gamma \vdash M : \sigma$ (pronounce this as 'Γ yields M in σ'). A particular Curry type assignment system depends on two parameters, the set $\mathbb{T}$ and the rules of type assignment. As an example we

now introduce the system $\lambda\!\!\rightarrow$-Curry.

5.1. DEFINITION. The set of *types* of $\lambda\!\!\rightarrow$, notation Type($\lambda\!\!\rightarrow$), is inductively defined as follows. We write $\mathbb{T} = \text{Type}(\lambda\!\!\rightarrow)$. Let $\mathbb{V} = \{\alpha, \alpha', \ldots\}$ be a set of *type variables*. It will be convenient to allow *type constants* for basic types such as Nat, Bool. Let $\mathbb{B}$ be such a collection. Then

$$\begin{aligned} \alpha \in \mathbb{V} &\Rightarrow \alpha \in \mathbb{T}, \\ \mathsf{B} \in \mathbb{B} &\Rightarrow \mathsf{B} \in \mathbb{T}, \\ \sigma, \tau \in \mathbb{T} &\Rightarrow (\sigma\!\rightarrow\!\tau) \in \mathbb{T} \quad \text{(function space types)}. \end{aligned}$$

For such definitions it is convenient to use the following abstract syntax to form $\mathbb{T}$.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\!\rightarrow\!\mathbb{T}$$

with

$$\mathbb{V} = \alpha \mid \mathbb{V}' \qquad \text{(type variables)}.$$

NOTATION. (i) If $\sigma_1, \ldots, \sigma_n \in \mathbb{T}$ then

$$\sigma_1\!\rightarrow\!\sigma_2\!\rightarrow\cdots\rightarrow\!\sigma_n$$

stands for

$$(\sigma_1\!\rightarrow\!(\sigma_2\!\rightarrow\cdots\rightarrow\!(\sigma_{n-1}\!\rightarrow\!\sigma_n)\cdot\cdot));$$

that is, we use association to the right.

(ii) $\alpha, \beta, \gamma, \ldots$ denote arbitrary type variables.

5.2. DEFINITION. (i) A *statement* is of the form $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$. This statement is pronounced as 'M in σ'. The type σ is the *predicate* and the term M is the *subject* of the statement.

(ii) A *basis* is a set of statements with only distinct (term) variables as subjects.

5.3. DEFINITION. Type *derivations* in the system $\lambda\!\!\rightarrow$ are built up from assumptions $x{:}\sigma$, using the following inference rules.

$$\frac{M : \sigma\!\rightarrow\!\tau \qquad N : \sigma}{MN : \tau} \qquad\qquad \frac{\begin{array}{c}\not{x : \sigma} \\ \vdots \\ M : \tau\end{array}}{\lambda x.M : \sigma\!\rightarrow\!\tau}$$

5.4. DEFINITION. (i) A statement $M : \sigma$ is *derivable from* a basis Γ, notation

$$\Gamma \vdash M : \sigma$$

(or $\Gamma \vdash_{\lambda\!\rightarrow} M : \sigma$ if we wish to stress the typing system) if there is a derivation of $M : \sigma$ in which all non-cancelled assumptions are in Γ.

(ii) We use $\vdash M : \sigma$ as shorthand for $\emptyset \vdash M : \sigma$.

5.5. Example. (i) Let $\sigma \in \mathbb{T}$. Then $\vdash \lambda f x.f(fx) : (\sigma\to\sigma)\to\sigma\to\sigma$, which is shown by the following derivation.

$$\dfrac{\dfrac{\dfrac{\cancel{f:\sigma\to\sigma}^{(2)} \qquad \dfrac{\cancel{f:\sigma\to\sigma}^{(2)} \qquad \cancel{x:\sigma}^{(1)}}{fx:\sigma}}{f(fx):\sigma}}{\lambda x.f(fx):\sigma\to\sigma}\,(1)}{\lambda f x.f(fx):(\sigma\to\sigma)\to\sigma\to\sigma}\,(2)$$

The indices (1) and (2) are bookkeeping devices that indicate at which application of a rule a particular assumption is being cancelled.

(ii) One has $\vdash \mathbf{K} : \sigma\to\tau\to\sigma$ for any $\sigma,\tau \in \mathbb{T}$, which is demonstrated as follows.

$$\dfrac{\dfrac{\cancel{x:\sigma}^{(1)}}{\lambda y.x:\tau\to\sigma}}{\lambda xy.x:\sigma\to\tau\to\sigma}\,(1)$$

(iii) Similarly one can show for all $\sigma \in \mathbb{T}$

$$\vdash \mathbf{I} : \sigma\to\sigma.$$

(iv) An example with a non-empty basis is the statement

$$y{:}\sigma \vdash \mathbf{I}y : \sigma.$$

Properties of $\lambda\to$

Several properties of type assignment in $\lambda\to$ are valid. The first one analyses how much of a basis is necessary in order to derive a type assignment.

5.6. Definition. Let $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ be a basis.

(i) Write $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\sigma_i = \Gamma(x_i)$. That is, Γ is considered as a partial function.

(ii) Let V_0 be a set of variables. Then $\Gamma \restriction V_0 = \{x{:}\sigma \mid x \in V_0 \,\&\, \sigma = \Gamma(x)\}$.

(iii) For $\sigma,\tau \in \mathbb{T}$ substitution of τ for α in σ is denoted by $\sigma[\alpha := \tau]$.

5.7. Basis Lemma. *Let Γ be a basis.*

(i) *If $\Gamma' \supseteq \Gamma$ is another basis, then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M : \sigma.$$

(ii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$.

(iii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \restriction \mathrm{FV}(M) \vdash M : \sigma$.

Proof. (i) By induction on the derivation of $M : \sigma$. Since such proofs will occur frequently we will spell it out in this simple situation in order to be shorter later on.

Case 1. $M : \sigma$ is $x{:}\sigma$ and is element of Γ. Then also $x{:}\sigma \in \Gamma'$ and hence $\Gamma' \vdash M : \sigma$.

Case 2. $M : \sigma$ is $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau \to \sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma' \vdash M_1 : (\tau \to \sigma)$ and $\Gamma' \vdash M_2 : \tau$. Hence $\Gamma' \vdash (M_1 M_2) : \sigma$.

Case 3. $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1 \to \sigma_2)$ and follows directly from $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$. By the variable convention it may be assumed that the bound variable x does not occur in $\mathrm{dom}(\Gamma')$. Then $\Gamma', x{:}\sigma_1$ is also a basis which extends $\Gamma, x{:}\sigma_1$. Therefore by the IH one has $\Gamma', x{:}\sigma_1 \vdash M_1 : \sigma_2$ and so $\Gamma' \vdash (\lambda x.M_1) : (\sigma_1 \to \sigma_2)$.

(ii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1 \to \sigma_2)$ and follows directly from $\Gamma, x{:}\sigma_1 \vdash M_1 : \sigma_2$. Let $y \in \mathrm{FV}(\lambda x.M_1)$, then $y \in \mathrm{FV}(M_1)$ and $y \not\equiv x$. By the IH one has $y \in \mathrm{dom}(\Gamma, x{:}\sigma_1)$ and therefore $y \in \mathrm{dom}(\Gamma)$.

(iii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau \to \sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma \restriction \mathrm{FV}(M_1) \vdash M_1 : (\tau \to \sigma)$ and $\Gamma \restriction \mathrm{FV}(M_2) \vdash M_2 : \tau$. By (i) it follows that $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash M_1 : (\tau \to \sigma)$and $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash M_2 : \tau$ and hence $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash (M_1 M_2) : \sigma$. □

The second property analyses how terms of a certain form get typed. It is useful among other things to show that certain terms have no types.

5.8. Generation Lemma. (i) $\Gamma \vdash x : \sigma \;\Rightarrow\; (x{:}\sigma) \in \Gamma$.
(ii) $\Gamma \vdash MN : \tau \;\Rightarrow\; \exists \sigma\, [\Gamma \vdash M : (\sigma \to \tau) \,\&\, \Gamma \vdash N : \sigma]$.
(iii) $\Gamma \vdash \lambda x.M : \rho \;\Rightarrow\; \exists \sigma, \tau\, [\Gamma, x{:}\sigma \vdash M : \tau \,\&\, \rho \equiv (\sigma \to \tau)]$.

Proof. By induction on the structure of derivations. □

5.9. Proposition (Typability of subterms). *Let M' be a subterm of M. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M' : \sigma' \quad \textit{for some } \Gamma' \textit{ and } \sigma'.$$

The moral is: if M has a type, i.e. $\Gamma \vdash M : \sigma$ for some Γ and σ, then every subterm has a type as well.

Proof. By induction on the generation of M. □

5.10. Substitution Lemma.
(i) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.
(ii) *Suppose* $\Gamma, x{:}\sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$. *Then* $\Gamma \vdash M[x := N] : \tau$.

Proof. (i) By induction on the derivation of $M : \sigma$.
(ii) By induction on the derivation showing $\Gamma, x{:}\sigma \vdash M : \tau$. □

The following result states that the set of $M \in \Lambda$ having a certain type in $\lambda{\to}$ is closed under reduction.

5.11. Subject Reduction Theorem. *Suppose $M \twoheadrightarrow_\beta M'$. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \vdash M' : \sigma.$$

PROOF. Induction on the generation of $\twoheadrightarrow_\beta$ using the Generation Lemma 5.8 and the Substitution Lemma 5.10. We treat the prime case, namely that $M \equiv (\lambda x.P)Q$ and $M' \equiv P[x := Q]$. Well, if

$$\Gamma \vdash (\lambda x.P)Q : \sigma$$

then it follows by the Generation Lemma that for some τ one has

$$\Gamma \vdash (\lambda x.P) : (\tau \to \sigma) \text{ and } \Gamma \vdash Q : \tau.$$

Hence once more by the Generation Lemma

$$\Gamma, x{:}\tau \vdash P : \sigma \text{ and } \Gamma \vdash Q : \tau$$

and therefore by the Substitution Lemma

$$\Gamma \vdash P[x := Q] : \sigma. \ \square$$

Terms having a type are not closed under expansion. For example,

$$\vdash \mathbf{I} : (\sigma \to \sigma), \text{ but } \not\vdash \mathbf{KI}\ (\lambda x.xx) : (\sigma \to \sigma).$$

See Exercise 5.1. One even has the following stronger failure of subject expansion, as is observed in van Bakel (1992).

5.12. OBSERVATION. There are $M, M' \in \Lambda$ and $\sigma, \sigma' \in \mathbb{T}$ such that $M' \twoheadrightarrow_\beta M$ and

$$\vdash M : \sigma, \qquad \vdash M' : \sigma',$$

but

$$\not\vdash M' : \sigma.$$

PROOF. Take $M \equiv \lambda xy.y$, $M' \equiv \mathbf{SK}$, $\sigma \equiv \alpha \to (\beta \to \beta)$ and $\sigma' \equiv (\beta \to \alpha) \to (\beta \to \beta)$; do Exercise 5.1. $\square$

All typable terms have a normal form. In fact, the so-called *strong normalization* property holds: if M is a typable term, then all reductions starting from M are finite.

Decidability of type assignment

For the system of type assignment several questions may be asked. Note that for $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ one has

$$\Gamma \vdash M : \sigma \ \Leftrightarrow \ \vdash (\lambda x_1{:}\sigma_1 \cdots \lambda x_n{:}\sigma_n.M) : (\sigma_1 \to \cdots \to \sigma_n \to \sigma),$$

therefore in the following one has taken $\Gamma = \emptyset$. Typical questions are

(1) Given M and σ, does one have $\vdash M : \sigma$?
(2) Given M, does there exist a σ such that $\vdash M : \sigma$?
(3) Given σ, does there exist an M such that $\vdash M : \sigma$?

These three problems are called *type checking*, *typability* and *inhabitation* respectively and are denoted by $M : \sigma?$, $M : ?$ and $? : \sigma$.

Type checking and typability are decidable. This can be shown using the following result, independently due to Curry (1969), Hindley (1969), and Milner (1978).

5.13. THEOREM. (i) *It is decidable whether a term is typable in* $\lambda\!\!\rightarrow$.

(ii) *If a term* M *is typable in* $\lambda\!\!\rightarrow$, *then* M *has a principal type scheme, i.e. a type* σ *such that every possible type for* M *is a substitution instance of* σ. *Moreover* σ *is computable from* M.

5.14. COROLLARY. *Type checking for* $\lambda\!\!\rightarrow$ *is decidable*.

PROOF. In order to check $M : \tau$ it suffices to verify that M is typable and that τ is an instance of the principal type of M. □

For example, a principal type scheme of **K** is $\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$.

Polymorphism

Note that in $\lambda\!\!\rightarrow$ one has

$$\vdash \mathbf{I} : \sigma\!\rightarrow\!\sigma \qquad \text{for all } \sigma \in \mathbb{T}.$$

In the polymorphic lambda calculus this quantification can be internalized by stating

$$\vdash \mathbf{I} : \forall\alpha.\alpha\!\rightarrow\!\alpha.$$

The resulting system is the *polymorphic* of *second-order* lambda calculus due to Girard (1972) and Reynolds (1974).

5.15. DEFINITION. The set of *types* of $\lambda 2$ (notation $\mathbb{T} = \mathrm{Type}(\lambda 2)$) is specified by the syntax

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\!\rightarrow\!\mathbb{T} \mid \forall\mathbb{V}.\mathbb{T}.$$

5.16. DEFINITION. The rules of type assignment are those of $\lambda\!\!\rightarrow$, plus

$$\frac{M : \forall\alpha.\sigma}{M : \sigma[\alpha := \tau]} \qquad \frac{M : \sigma}{M : \forall\alpha.\sigma}$$

In the latter rule, the type variable α may not occur free in any assumption on which the premiss $M : \sigma$ depends.

5.17. EXAMPLE. (i) $\vdash \mathbf{I} : \forall\alpha.\alpha\!\rightarrow\!\alpha$.

(ii) Define $\mathrm{Nat} \equiv \forall\alpha.(\alpha\!\rightarrow\!\alpha)\!\rightarrow\!\alpha\!\rightarrow\!\alpha$. Then for the Church numerals $\boldsymbol{c}_n \equiv \lambda f x.f^n(x)$ we have $\vdash \boldsymbol{c}_n : \mathrm{Nat}$.

The following is due to Girard (1972).

5.18. THEOREM. (i) *The Subject Reduction property holds for* $\lambda 2$.

(ii) $\lambda 2$ *is strongly normalizing.*

Typability in $\lambda 2$ is *not* decidable; see Wells (1994).

Exercises

5.1. (i) Give a derivation of
$$\vdash \mathbf{SK} : (\alpha\to\beta)\to(\alpha\to\alpha).$$
(ii) Give a derivation of
$$\vdash \mathbf{KI} : \beta\to(\alpha\to\alpha).$$
(iii) Show that $\not\vdash \mathbf{SK} : (\alpha\to\beta\to\beta)$.
(iv) Find a common β-reduct of **SK** and **KI**. What is the most general type for this term?

5.2. Show that $\lambda x.xx$ and $\mathbf{KI}(\lambda x.xx)$ have no type in $\lambda\to$.

5.3. Find the most general types (if they exist) for the following terms.
(i) $\lambda xy.xyy$.
(ii) **SII**.
(iii) $\lambda xy.y(\lambda z.z(yx))$.

5.4. Find terms $M, N \in \Lambda$ such that the following hold in $\lambda\to$.
(i) $\vdash M : (\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)$.
(ii) $\vdash N : (((\alpha\to\beta)\to\beta)\to\beta)\to(\alpha\to\beta)$.

5.5. Find types in $\lambda 2$ for the terms in the exercises 5.2 and 5.3.

Chapter 6

Extensions

In Chapter 3 we have seen that all computable functions can be expressed in the lambda calculus. For reasons of efficiency, reliability and convenience this language will be extended. The set of λ-terms Λ will be extended with constants. Some of the constants are selected to represent primitive data (such as numbers) and operations on these (such as addition). Some new reduction rules (the so called δ-rules) are introduced to express the operational semantics of these operations. Even if these constants and operations can be implemented in the lambda calculus, it is worthwhile to have primitive symbols for them. The reason is that in an implementation of the lambda calculus addition of the Church numerals runs less efficient than the usual implementation in hardware of addition of binary represented numbers. Having numerals and addition as primitives therefore creates the possibility to interprete these efficiently.

From now on we allow constants in λ-terms. Let $\mathbb{C}$ be a set of constants.

6.1. Definition. The set of *lambda terms with constants*, notation $\Lambda(\mathbb{C})$, is defined inductively as follows.

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

This definition given as an abstract syntax is as follows.

$$\Lambda(\mathbb{C}) ::= \mathbb{C} \mid V \mid \Lambda(\mathbb{C})\,\Lambda(\mathbb{C}) \mid \lambda V\,\Lambda(\mathbb{C}).$$

6.2. Definition (δ-reduction). Let $X \subseteq \Lambda(\mathbb{C})$ be a set of closed normal forms. Usually we take $X \subseteq \mathbb{C}$. Let $f : X^k \to \Lambda$ be an 'externally defined' function. In order to represent f, a so-called δ-rule may be added to the λ-calculus. This is done as follows.

(1) A special constant in $\mathbb{C}$ is selected and is given some name, say $\boldsymbol{\delta}$ $(= \boldsymbol{\delta}_f)$.

(2) The following contraction rules are added to those of the λ-calculus:

$$\boldsymbol{\delta} M_1 \cdots M_k \to f(M_1, \ldots, M_k),$$

for $M_1, \ldots, M_k \in X$.

Note that for a given function f this is not one contraction rule but in fact a rule *schema*. The resulting extension of the λ-calculus is called $\lambda\delta$. The corresponding notion of (one step) reduction is denoted by $(\to_{\beta\delta})\ \twoheadrightarrow_{\beta\delta}$.

So δ-reduction is not an absolute notion, but depends on the choice of f.

6.3. THEOREM (G. Mitschke). *Let f be a function on closed normal forms. Then the resulting notion of reduction $\twoheadrightarrow_{\beta\delta}$ satisfies the Church-Rosser property.*

PROOF. Follows from Theorem 15.3.3 in Barendregt (1984). □

The notion of normal form generalises to $\beta\delta$-normal form. So does the concept of leftmost reduction. The $\beta\delta$-normalforms can be found by a leftmost reduction (notation $\twoheadrightarrow_{\ell\beta\delta}$).

6.4. THEOREM. *If $M \twoheadrightarrow_{\beta\delta} N$ and N is in $\beta\delta$-nf, then $M \twoheadrightarrow_{\ell\beta\delta} N$.*

PROOF. Analogous to the proof of the theorem for β-normal forms (4.22). □

6.5. EXAMPLE. One of the first versions of a δ-rule is in Church (1941). Here X is the set of all closed normal forms and for $M, N \in X$ we have

$$\begin{aligned} \boldsymbol{\delta}_{\mathrm{C}} MN &\to \mathsf{true}, && \text{if } M \equiv N; \\ \boldsymbol{\delta}_{\mathrm{C}} MN &\to \mathsf{false}, && \text{if } M \not\equiv N. \end{aligned}$$

Another possible set of δ-rules is for the Booleans.

6.6. EXAMPLE. The following constants are selected in $\mathbb{C}$.

$$\boldsymbol{true}, \boldsymbol{false}, \boldsymbol{not}, \boldsymbol{and}, \boldsymbol{ite} \text{ (for } \textit{if then else}).$$

The following δ-rules are introduced.

$$\begin{aligned} \boldsymbol{not\ true} &\to \boldsymbol{false}; \\ \boldsymbol{not\ false} &\to \boldsymbol{true}; \\ \boldsymbol{and\ true\ true} &\to \boldsymbol{true}; \\ \boldsymbol{and\ true\ false} &\to \boldsymbol{false}; \\ \boldsymbol{and\ false\ true} &\to \boldsymbol{false}; \\ \boldsymbol{and\ false\ false} &\to \boldsymbol{false}; \\ \boldsymbol{ite\ true} &\to \mathsf{true} \quad (\equiv \lambda xy.x); \\ \boldsymbol{ite\ false} &\to \mathsf{false} \quad (\equiv \lambda xy.x). \end{aligned}$$

It follows that

$$\begin{aligned} \boldsymbol{ite\ true}\, x\, y &\twoheadrightarrow x, \\ \boldsymbol{ite\ false}\, x\, y &\twoheadrightarrow y. \end{aligned}$$

Now we introduce as δ-rules some operations on the set of integers

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

6.7. Example. For each $n \in \mathbb{Z}$ a constant in $\mathbb{C}$ is selected and given the name $\boldsymbol{n}$. (We will express this as follows: for each $n \in \mathbb{Z}$ a constant $\boldsymbol{n} \in \mathbb{C}$ is chosen.) Moreover the following constants in $\mathbb{C}$ are selected:

$$\boldsymbol{plus}, \boldsymbol{minus}, \boldsymbol{times}, \boldsymbol{divide}, \boldsymbol{equal}, \boldsymbol{error}.$$

Then we introduce the following δ-rules (schemes). For $m, n \in \mathbb{Z}$

$$\begin{aligned}
\boldsymbol{plus}\,\boldsymbol{n}\,\boldsymbol{m} &\to \boldsymbol{n} + \boldsymbol{m};\\
\boldsymbol{minus}\,\boldsymbol{n}\,\boldsymbol{m} &\to \boldsymbol{n} - \boldsymbol{m};\\
\boldsymbol{times}\,\boldsymbol{n}\,\boldsymbol{m} &\to \boldsymbol{n} * \boldsymbol{m};\\
\boldsymbol{divide}\,\boldsymbol{n}\,\boldsymbol{m} &\to \boldsymbol{n} \div \boldsymbol{m}, \quad \text{if } m \neq 0;\\
\boldsymbol{divide}\,\boldsymbol{n}\,\boldsymbol{0} &\to \boldsymbol{error};\\
\boldsymbol{equal}\,\boldsymbol{n}\,\boldsymbol{n} &\to \boldsymbol{true};\\
\boldsymbol{equal}\,\boldsymbol{n}\,\boldsymbol{m} &\to \boldsymbol{false}, \quad \text{if } n \neq m.
\end{aligned}$$

We may add rules like

$$\boldsymbol{plus}\,\boldsymbol{n}\,\boldsymbol{error} \to \boldsymbol{error}.$$

Similar δ-rules can be introduced for the set of reals.

Again another set of δ-rules is concerned with characters.

6.8. Example. Let Σ be some linearly ordered alphabet. For each symbol $s \in \Sigma$ we choose a constant 's' $\in \mathbb{C}$. Moreover we choose two constants $\boldsymbol{\delta}_{\leq}$ and $\boldsymbol{\delta}_{=}$ in $\mathbb{C}$ and formulate the following δ-rules.

$$\begin{aligned}
\boldsymbol{\delta}_{\leq}\text{'}s_1\text{''}s_2\text{'} &\to \boldsymbol{true}, && \text{if } s_1 \text{ precedes } s_2 \text{ in the ordering of } \Sigma;\\
\boldsymbol{\delta}_{\leq}\text{'}s_1\text{''}s_2\text{'} &\to \boldsymbol{false}, && \text{otherwise.}\\
\boldsymbol{\delta}_{=}\text{'}s_1\text{''}s_2\text{'} &\to \boldsymbol{true}, && \text{if } s_1 = s_2;\\
\boldsymbol{\delta}_{=}\text{'}s_1\text{''}s_2\text{'} &\to \boldsymbol{false}, && \text{otherwise.}
\end{aligned}$$

It is also possible to represent 'multiple valued' functions F by putting as δ-rule

$$\boldsymbol{\delta n} \to \boldsymbol{m}, \quad \text{provided that } F(n) = m.$$

Of course the resulting $\lambda\delta$-calculus does not satisfy the Church-Rosser theorem and can be used to deal with non-deterministic computations. We will not pursue this possibility, however.

We can extend the type assignment system $\lambda{\to}$ to deal with constants by adding typing *axioms* of the form

$$\boldsymbol{C} : \sigma.$$

For the system with integers this would result in the following. Let $\mathsf{Z}, \mathsf{B} \in \mathbb{B}$ be basic type constants (with intended interpretation $\mathbb{Z}$ and booleans, respectively). Then one adds the following typing axioms to $\lambda{\to}$.

$$\begin{gathered}
\boldsymbol{true} : \mathsf{B}, \quad \boldsymbol{false} : \mathsf{B}, \\
\boldsymbol{not} : \mathsf{B}{\to}\mathsf{B}, \quad \boldsymbol{and} : \mathsf{B}{\to}\mathsf{B}{\to}\mathsf{B}, \\
\boldsymbol{n} : \mathsf{Z}, \quad \boldsymbol{error} : \mathsf{Z}, \\
\boldsymbol{plus} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}, \quad \boldsymbol{minus} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}, \quad \boldsymbol{times} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}, \quad \boldsymbol{divide} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}, \\
\boldsymbol{equal} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{B}.
\end{gathered}$$

6.9. Example. $\vdash \lambda xy.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}$, as is shown by the following derivation.

$$\dfrac{\dfrac{\dfrac{\boldsymbol{times} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z} \qquad \cancel{x : \mathsf{Z}}^{(2)}}{\boldsymbol{times}\, x : \mathsf{Z}{\to}\mathsf{Z}} \qquad \dfrac{\dfrac{\boldsymbol{plus} : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z} \qquad \cancel{x : \mathsf{Z}}^{(2)}}{\boldsymbol{plus}\, x : \mathsf{Z}{\to}\mathsf{Z}} \qquad \cancel{y : \mathsf{Z}}^{(1)}}{\boldsymbol{plus}\, xy : \mathsf{Z}}}{\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}}}{\lambda y.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}{\to}\mathsf{Z}}\,{}^{(1)}}{\lambda xy.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}{\to}\mathsf{Z}{\to}\mathsf{Z}}\,{}^{(2)}$$

The Strong Normalization property for (plain) $\lambda{\to}$ implies that not all recursive functions are definable in the system. The same holds for the above $\lambda\delta$-calculus with integers. The following system of type assignment is such that all computable functions are representable by a typed term. Indeed, the system also assigns types to non-normalizing terms by introducing a primitive fixedpoint combinator $\boldsymbol{Y}$ having type $(\sigma{\to}\sigma){\to}\sigma$ for every σ.

6.10. Definition. (i) The *$\lambda\boldsymbol{Y}\delta$-calculus* is an extension of the $\lambda\delta$-calculus in which there is a constant $\boldsymbol{Y}$ with reduction rule

$$\boldsymbol{Y}f \to f(\boldsymbol{Y}f).$$

(ii) Type assignment to $\lambda\boldsymbol{Y}\delta$-terms is defined by adding the axioms

$$\boldsymbol{Y} : (\sigma{\to}\sigma){\to}\sigma$$

for each $\sigma \in \mathbb{T}$. The resulting system is denoted by $\lambda\boldsymbol{Y}\delta{\to}$.

Because of the presence of $\boldsymbol{Y}$, not all terms have a normal form. Without proof we state the following.

6.11. Theorem. (i) *The $\lambda\boldsymbol{Y}\delta$-calculus satisfies the Church-Rosser property.*
(ii) *If a term in the $\lambda\boldsymbol{Y}\delta$-calculus has a normal form, then it can be found using leftmost reduction.*
(iii) *The Subject Reduction property holds for $\lambda\boldsymbol{Y}\delta{\to}$.*

6.12. THEOREM. *All computable functions can be represented in the $\lambda \boldsymbol{Y} \delta$-calculus by a term typable in $\lambda \boldsymbol{Y} \delta{\rightarrow}$.*

PROOF. The construction uses the primitive numerals $\boldsymbol{n}$. If we take $\mathsf{S}^+_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{plus}\, x\, \mathbf{1}$, $\mathsf{P}^-_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{minus}\, x\, \mathbf{1}$, and $\mathsf{Zero}_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{equal}\, x\, \mathbf{0}$, then the proof of Theorem 3.13 can be imitated using $\boldsymbol{Y}$ instead of the fixedpoint combinator $\mathbf{Y}$. The types for the functions defined using $\boldsymbol{Y}$ are natural. □

One could also add $\boldsymbol{Y}$ to the system $\lambda 2$ using the (single) axiom

$$\boldsymbol{Y} : \forall\alpha.(\alpha{\rightarrow}\alpha){\rightarrow}\alpha.$$

Exercises

6.1. Let $\boldsymbol{k}_n$ be defined by $\boldsymbol{k}_0 \equiv \mathsf{I}$ and $\boldsymbol{k}_{n+1} \equiv \mathsf{K}(\boldsymbol{k}_n)$. Show that on the $\boldsymbol{k}_n$ the recursive functions can be represented by terms in the $\lambda\delta_{\mathrm{C}}$-calculus.

6.2. Write down a $\lambda\delta$-term F in the system of Example 6.7 such that

$$F\boldsymbol{n} \twoheadrightarrow \boldsymbol{n!} + \boldsymbol{n}.$$

6.3. Write down a $\lambda\delta$-term F in the system of Example 6.8 such that for $s_1, s_2, t_1, t_2 \in \Sigma$ we have

$$\begin{array}{lll} F[\text{`}s_1\text{'},\text{`}t_1\text{'}][\text{`}s_2\text{'},\text{`}t_2\text{'}] & \twoheadrightarrow \boldsymbol{true}, & \text{if } (s_1,t_1) \text{ precedes } (s_2,t_2) \text{ in the} \\ & & \text{lexicographical ordering of } \Sigma\times\Sigma; \\ & \twoheadrightarrow \boldsymbol{false}, & \text{otherwise.} \end{array}$$

6.4. Give suitable typing axioms (in $\lambda{\rightarrow}$ and $\lambda 2$) for the constants in Example 6.6.

Chapter 7

Reduction Systems

In this chapter we consider some alternative models of computation based on rewriting. The objects in these models are terms built up from *constants* with *arity* in $\mathbb{N}$ and variables, using application.

7.1. Definition. Let $\mathcal{C}$ be a set of constants. The set of *terms over* $\mathcal{C}$ (notation $\mathcal{T} = \mathcal{T}(\mathcal{C})$) is defined as follows.

$$\begin{aligned} x \in V &\Rightarrow x \in \mathcal{T}, \\ \boldsymbol{C} \in \mathcal{C}, t_1 \ldots, t_k \in \mathcal{T} &\Rightarrow \boldsymbol{C}(t_1, \ldots, t_n) \in \mathcal{T}, \end{aligned}$$

where $n = \text{arity}(\boldsymbol{C})$.

Recursive programming schemes

The simplest reduction systems are *recursive programming schemes* (RPS).

The general form of an RPS has as language the terms $\mathcal{T}(\mathcal{C})$. On these a reduction relation is defined as follows.

$$\begin{aligned} \boldsymbol{C}_1(x_1, \ldots, x_{n_1}) &\to t_1, \\ &\vdots \\ \boldsymbol{C}_k(x_1, \ldots, x_{n_k}) &\to t_k, \end{aligned}$$

where $n_i = \text{arity}(\boldsymbol{C}_i)$. Here we have

(1) The $\boldsymbol{C}$'s are all different constants.
(2) The free variables in t_i are among the $x_1, \ldots, x_{n_i}$.
(3) In the t's there may be arbitrary $\boldsymbol{C}$'s.

For example, the system

$$\begin{aligned} \boldsymbol{C}(x, y) &\to \boldsymbol{D}(\boldsymbol{C}(x, x), y), \\ \boldsymbol{D}(x, y) &\to \boldsymbol{C}(x, \boldsymbol{D}(x, y)) \end{aligned}$$

is an RPS.

The λ-calculus is powerful enough to 'implement' all these RPS's. We can find λ-terms with the specified reduction behaviour.

7.2. THEOREM. *Each RPS can be represented in λ-calculus. For example (see above), there are terms* **C** *and* **D** *such that*

$$\begin{aligned} \mathbf{C}xy &\twoheadrightarrow_\beta \mathbf{D}(\mathbf{C}xx)y, \\ \mathbf{D}xy &\twoheadrightarrow_\beta \mathbf{C}x(\mathbf{D}xy). \end{aligned}$$

PROOF. By the reducing variant 4.27 of the Multiple Fixedpoint Theorem. □

Without proof we mention the following.

7.3. THEOREM. *Every RPS satisfies the Church-Rosser theorem.*

Term rewrite systems

More general than the RPS's are the so called *term rewrite systems* (TRS's), which use *pattern matching* in function definitions. A typical example is

$$\begin{aligned} \boldsymbol{A}(\mathbf{0}, y) &\to y, \\ \boldsymbol{A}(\boldsymbol{S}(x), y) &\to \boldsymbol{S}(\boldsymbol{A}(x, y)). \end{aligned}$$

Then, for example, $\boldsymbol{A}(\boldsymbol{S}(\mathbf{0}), \boldsymbol{S}(\boldsymbol{S}(\mathbf{0}))) \twoheadrightarrow \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{S}(\mathbf{0})))$.

The difference with RPS's is that in a TRS the arguments of a rewrite rule may have some structure. A constant in a TRS that does not have a contraction rule (i.e. no rewrite rule starts with that constant) is called a *constructor*. The other constants are called *functions*.

Not all TRS's satisfy the Church-Rosser property. Consider the system

$$\begin{aligned} \boldsymbol{A}(x) &\to \boldsymbol{B}, \\ \boldsymbol{A}(\boldsymbol{B}) &\to \boldsymbol{C}. \end{aligned}$$

Then $\boldsymbol{A}(\boldsymbol{B})$ reduces both to $\boldsymbol{B}$ and to $\boldsymbol{C}$. It is said that the two rules *overlap*. The following rule overlaps with itself:

$$\boldsymbol{D}(\boldsymbol{D}(x)) \to \boldsymbol{E}.$$

Then $\boldsymbol{D}(\boldsymbol{D}(\boldsymbol{D}(\boldsymbol{D})))$ reduces to $\boldsymbol{E}$ and to $\boldsymbol{D}(\boldsymbol{E})$.

See Klop (1992) for a survey and references on TRS's.

Combinatory logic (CL) is a reduction system related to λ-calculus. Terms in CL consist of (applications of) constants $\boldsymbol{I}$, $\boldsymbol{K}$, $\boldsymbol{S}$ and variables, without arity restrictions. The contraction rules are

$$\begin{aligned} \boldsymbol{I}x &\to x, \\ \boldsymbol{K}xy &\to x, \\ \boldsymbol{S}xyz &\to xz(yz). \end{aligned}$$

(Note that $\boldsymbol{KI}$ is a nf.) Then $\boldsymbol{KII} \to \boldsymbol{I}$, and $\boldsymbol{SII}(\boldsymbol{SII})$ has no normal form. This CL can be represented as a TRS by considering $\boldsymbol{I}, \boldsymbol{K}, \boldsymbol{S}$ as (0-ary) constructors, together with a function $\boldsymbol{Ap}$ with arity 2, as follows.

$$\begin{aligned} \boldsymbol{Ap}(\boldsymbol{I}, x) &\to x, \\ \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{K}, x), y) &\to x, \\ \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{S}, x), y), z) &\to \boldsymbol{Ap}(\boldsymbol{Ap}(x, z), \boldsymbol{Ap}(y, z)). \end{aligned}$$

The CL-term $\boldsymbol{SII}(\boldsymbol{SII})$ is translated into $\underline{\boldsymbol{\Omega}} \equiv \boldsymbol{Ap}(\underline{\boldsymbol{\omega}}, \underline{\boldsymbol{\omega}})$ where $\underline{\boldsymbol{\omega}} \equiv \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{S}, \boldsymbol{I}), \boldsymbol{I})$.

The Normalization Theorem does not extend to TRS's. Consider the above TRS-version of CL, together with the rules

$$\begin{aligned} \boldsymbol{or}(x, \boldsymbol{true}) &\rightarrow \boldsymbol{true}, \\ \boldsymbol{or}(\boldsymbol{true}, x) &\rightarrow \boldsymbol{true}, \\ \boldsymbol{or}(\boldsymbol{false}, \boldsymbol{false}) &\rightarrow \boldsymbol{false}. \end{aligned}$$

The expression

$$\boldsymbol{or}(A, B)$$

can, in general, not be normalized by contracting always the leftmost redex. In fact A and B have to be evaluated in parallel. Consider e.g. the terms

$$\boldsymbol{or}(\underline{\boldsymbol{\Omega}}, \boldsymbol{Ap}(\boldsymbol{I}, \boldsymbol{true}))$$

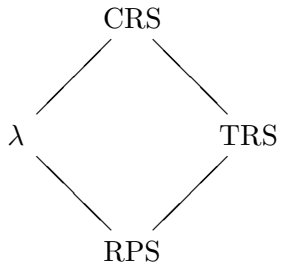
and

$$\boldsymbol{or}(\boldsymbol{Ap}(\boldsymbol{I}, \boldsymbol{true}), \underline{\boldsymbol{\Omega}}).$$

Therefore this system is called *non-sequential*.

Combinatory reduction systems

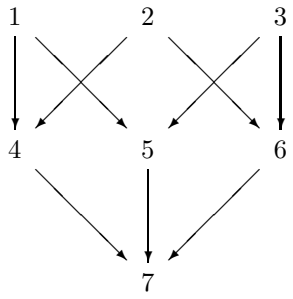
Even more general than TRS's are the combinatory reduction systems (CRS) introduced in Klop (1980). These are TRS's together with arbitrary variable binding operations. We have in fact



Exercises

7.1. (Toyama et al. (1989a), see also (1989b)) A TRS is called *strongly normalizing* (SN) if there is no term that has an infinite reduction path. So (the TRS version of) CL($\boldsymbol{S}, \boldsymbol{K}$) is not SN, but CL($\boldsymbol{I}, \boldsymbol{K}$) (with the obvious reduction rules) is. Define the following two TRS's.
$\mathcal{R}_1$:

$$\begin{aligned} F(4, 5, 6, x) &\rightarrow F(x, x, x, x), \\ F(x, y, z, w) &\rightarrow 7, \end{aligned}$$



$\mathcal{R}_2$:

$$\begin{aligned} G(x,x,y) &\to x, \\ G(x,y,x) &\to x, \\ G(y,x,x) &\to x. \end{aligned}$$

Show that both $\mathcal{R}_1$ and $\mathcal{R}_2$ are SN, but the union $\mathcal{R}_1 \cup \mathcal{R}_2$ is not.

Bibliography

Abramsky, S., D.M. Gabbay and T.S.E. Maibaum (eds.) (1992). *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.

van Bakel, S.J. (1992). Complete restrictions of the intersection type discipline, *Theoretical Computer Science* **102**, pp. 135–163.

Barendregt, H.P. (1976). A global representation of the recursive functions in the lambda calculus, *Theoretical Computer Science* **3**, pp. 225–242.

Barendregt, H.P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic 103, second, revised edition, North-Holland, Amsterdam.

Barendregt, H.P. (1990). Functional programming and lambda calculus, *in:* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. II, Elsevier/MIT Press.

Barendregt, H.P. (1992). Lambda calculi with types, *in:* Abramsky et al. (1992), pp. 117–309.

de Bruijn, N.G. (1970). The mathematical language AUTOMATH, its usage and some of its extensions, *in:* M. Laudet, D. Lacombe and M. Schuetzenberger (eds.), *Symposium on Automatic Demonstration*, INRIA, Versailles, Lecture Notes in Computer Science 125, Springer-Verlag, Berlin, pp. 29–61. Also in Nederpelt et al. (1994).

de Bruijn, N.G. (1980). A survey of the AUTOMATH project, *in:* Hindley and Seldin (1980), pp. 580–606.

Church, A. (1936). An unsolvable problem of elementary number theory, *American Journal of Mathematics* **58**, pp. 354–363.

Church, A. (1940). A formulation of the simple theory of types, *Journal of Symbolic Logic* **5**, pp. 56–68.

Church, A. (1941). *The Theory of Lambda Conversion*, Princeton University Press.

Curry, H.B. (1934). Functionality in combinatory logic, *Proceedings of the National Academy of Science USA* **20**, pp. 584–590.

Curry, H.B. (1969). Modified basic functionality in combinatory logic, *Dialectica* **23**, pp. 83–92.

Curry, H.B. and R. Feys (1958). *Combinatory Logic*, Vol. I, North-Holland, Amsterdam.

Curry, H.B., J.R. Hindley and J.P. Seldin (1972). *Combinatory Logic*, Vol. II, North-Holland, Amsterdam.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*, Dissertation, Université Paris VII.

Hindley, J.R. (1969). The principal typescheme of an object in combinatory logic, *Transactions of the American Mathematical Society* **146**, pp. 29–60.

Hindley, J.R. and J.P. Seldin (eds.) (1980). *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York.

Howard, W.A. (1980). The formulae-as-types notion of construction, *in:* Hindley and Seldin (1980), pp. 479–490.

Kleene, S.C. (1936). λ-definability and recursiveness, *Duke Mathematical Journal* **2**, pp. 340–353.

Klop, J.W. (1980). *Combinatory Reduction Systems*, Dissertation, Utrecht University. CWI Tract, Amsterdam.

Klop, J.W. (1992). Term rewrite systems, *in:* Abramsky et al. (1992).

Läuchli, H. (1970). An abstract notion of realizability for which intuitionistic predicate logic is complete, *in:* G. Myhill, A. Kino and R. Vesley (eds.), *Intuitionism and Proof Theory: Proceedings of the Summer School Conference*, Buffalo, New York, North-Holland, Amsterdam, pp. 227–234.

Milner, R. (1978). A theory of type polymorphism in programming, *Journal of Computer and Systems Analysis* **17**, pp. 348–375.

Milner, R. (1984). A proposal for standard ML, *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, pp. 184–197.

Nederpelt, R.P., J.H. Geuvers and R.C. de Vrijer (eds.) (1994). *Selected Papers on Automath*, Studies in Logic 133, North-Holland, Amsterdam.

Prawitz, D. (1965). *Natural Deduction: A Proof-Theoretical Study*, Almqvist and Wiksell, Stockholm.

Reynolds, J.C. (1974). Towards a theory of type structure, *Colloque sur la Démonstration*, Paris, Lecture Notes in Computer Science 19, Springer-Verlag, Berlin, pp. 408–425.

Schönfinkel, M. (1924). Über die Bausteinen der mathematische Logik, *Mathematische Annalen* **92**, pp. 305–316.

Schwichtenberg, H. (1977). Proof theory: applications of cut-elimination, *in:* J. Barwise (ed.), *Handbook of Mathematical Logic*, North-Holland, Amsterdam, pp. 867–895.

Toyama, Y., J.W. Klop and H.P. Barendregt (1989a). Termination for the direct sum of left-linear term rewriting systems, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, Chapel Hill, Lecture Notes in Computer Science 355, Springer-Verlag, Berlin, pp. 477–491.

Toyama, Y., J.W. Klop and H.P. Barendregt (1989b). Termination for the direct sum of left-linear term rewriting systems, *Technical Report CS-R8923*, Centre for Mathematics and Computer Science (CWI), Amsterdam.

Turing, A.M. (1936/7). On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* **42**, pp. 230–265.

Turing, A.M. (1937). Computability and λ-definability, *Journal of Symbolic Logic* **2**, pp. 153–163.

Wells, J.B. (1994). Typability and type-checking in the second-order λ-calculus are equivalent and undecidable, *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, Paris, France, IEEE Computer Society Press, pp. 176–185.