also Pavlović (1990). For the semantics of the Curry systems see Hindley (1982), (1983) and Coppo (1985). A later volume of this handbook will contain a chapter on the semantics of typed lambda calculi.

Barendregt and Hemerik (1990) and Barendregt (1991) are introductory versions of this chapter. Books including material on typed lambda calculus are Girard *et al.* (1989) (treats among other things semantics of the Church version of $\lambda 2$), Hindley and Seldin (1986) (Curry and Church versions of $\lambda\!\rightarrow$), Krivine (1990) (Curry versions of $\lambda 2$ and $\lambda\cap$), Lambek and Scott (1986) (categorical semantics of $\lambda\!\rightarrow$) and the forthcoming Barendregt and Dekkers (199-) and Nerode and Odifreddi (199-).

Section 2 of this chapter is an introduction to type-free lambda-calculus and may be skipped if the reader is familiar with this subject. Section 3 explains in more detail the Curry and Church approach to lambda calculi with types. Section 4 is about the Curry systems and Section 5 is about the Church systems. These two sections can be read independently of each other.

2 Type-free lambda calculus

The introduction of the type-free lambda calculus is necessary in order to define the system of Curry type assignment on top of it. Moreover, although the Church style typed lambda calculi can be introduced directly, it is nevertheless useful to have some knowledge of the type-free lambda calculus. Therefore this section is devoted to this theory. For more information see Hindley and Seldin [1986] or Barendregt [1984].

2.1 The system

In this chapter the type-free lambda calculus will be called 'λ-calculus' or simply λ. We start with an informal description.

Application and abstraction

The λ-calculus has two basic operations. The first one is application. The expression

$$F.A$$

(usually written as FA) denotes the data F considered as algorithm applied to A considered as input. The theory λ 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 intuitive map

$$x \mapsto M[x],$$

i.e. to x one assigns $M[x]$. The variable x does not need to occur actually in M. In that case $\lambda x.M[x]$ is a constant function with value M.

Application and abstraction work together in the following intuitive formula:

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

That is, $(\lambda x.x^2 + 1)3$ denotes the function $x \mapsto x^2 + 1$ applied to the argument 3 giving $3^2 + 1$ (which is 10). In general we have

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

This last equation is preferably written as

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

where $[x := N]$ denotes substitution of N for x. This equation is called β-conversion. It is remarkable that although it 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. For example, 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:

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

In integral 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, obtaining $\int_b^a f(7, y)d7$; but substitution for y does make sense, obtaining $\int_b^a f(x, 7)dx$.

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. For example, $\lambda x.x$ 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. Equations like $\lambda x.x \equiv \lambda y.y$ are usually called α-conversion.

Functions of several 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

$$(Fx)y = F_x y = f(x,y). \tag{1}$$

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

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

The equation (1) then becomes

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

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

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

Then we have for F defined above

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

and (1) becomes

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

For n arguments we have

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

by using (β) n times. This last equation becomes in convenient vector notation

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

more generally one has

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

Now we give the formal description of the λ-calculus.

Definition 2.1.1. The set of λ-*terms*, notation Λ, is built up from an infinite set of variables $V = \{v, v', v'', \dots\}$ using application and (function) abstraction:

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

Using abstract syntax one may write the following.

$$\begin{array}{l} V ::= v \mid V' \\ \Lambda ::= V \mid (\Lambda\Lambda) \mid (\lambda V \Lambda) \end{array}$$

Example 2.1.2. The following are λ-terms:

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

Convention 2.1.3.

1. $x, y, z, \dots$ denote arbitrary variables;
   $M, N, L, \dots$ denote arbitrary λ-terms.
2. As already mentioned informally, the following abbreviations are used:

   $$FM_1 \dots M_n \text{ stands for } (..((FM_1)M_2) \dots M_n)$$

   and

   $$\lambda x_1 \cdots x_n.M \text{ stands for } (\lambda x_1(\lambda x_2(\dots(\lambda x_n(M))..))).$$
3. Outermost parentheses are not written.

Using this convention, the examples in 2.1.2 now may be written as follows:

$$\begin{array}{l} x; xz; \lambda x.xz; \\ (\lambda x.xz)y; \\ (\lambda y.(\lambda x.xz)y)w. \end{array}$$

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

Notation 2.1.4. $M \equiv N$ denotes that M and N are the same term or can be obtained from each other by renaming bound variables. For example,

$$\begin{array}{rcl} (\lambda x.x)z & \equiv & (\lambda x.x)z; \\ (\lambda x.x)z & \equiv & (\lambda y.y)z; \\ (\lambda x.x)z & \not\equiv & (\lambda x.y)z. \end{array}$$

Definition 2.1.5.

1. The set of *free variables* of M, (notation $FV(M)$), is defined inductively as follows:

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

2. M is a *closed* λ*-term* (or *combinator*) if $FV(M) = \emptyset$. The set of closed λ-terms is denoted by Λ^0.

3. The result of *substitution* of N for (the free occurrences of) x in M, notation $M[x := N]$, is defined as follows: Below $x \not\equiv y$.

$$\begin{array}{rcl} x[x := N] & \equiv & N; \\ y[x := N] & \equiv & y; \\ (PQ)[x := N] & \equiv & (P[x := N])(Q[x := N]); \\ (\lambda y.P)[x := N] & \equiv & \lambda y.(P[x := N]), \text{ provided } y \not\equiv x; \\ (\lambda x.P)[x := N] & \equiv & (\lambda x.P). \end{array}$$

In the λ-term

$$y(\lambda xy.xyz)$$

y and z occur as free variables; x and y occur as bound variables. The term $\lambda xy.xxy$ is closed.

Names of bound variables will be always chosen such that they differ from the free ones in a term. So one writes $y(\lambda xy'.xy'z)$ for $y(\lambda xy.xyz)$. This so-called 'variable convention' makes it possible to use substitution for the λ-calculus without a proviso on free and bound variables.

Proposition 2.1.6 (Substitution lemma). *Let $M, N, L \in \Lambda$. Suppose $x \not\equiv y$ and $x \notin FV(L)$. Then*

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

Proof. By induction on the structure of M. ■ ▮

Now we introduce the λ-calculus as a formal theory of equations between λ-terms.

Definition 2.1.7.

1. The principal axiom scheme of the λ-calculus is

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

   for all $M, N \in \Lambda$. This is called *β-conversion.*

2. There are also the 'logical' axioms and rules:

$$\begin{array}{rcl} & & M = M; \\ M = N & \Rightarrow & N = M; \\ M = N, N = L & \Rightarrow & M = L; \\ M = M' & \Rightarrow & MZ = M'Z; \\ M = M' & \Rightarrow & ZM = ZM'; \\ M = M' & \Rightarrow & \lambda x.M = \lambda x.M'. \end{array} \tag{ξ}$$

3. If $M = N$ is provable in the λ-calculus, then we write $\lambda \vdash M = N$ or sometimes just $M = N$.

Remarks 2.1.8.

1. 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 of *α-conversion*.

$$\lambda x.M = \lambda y.M[x := y], \tag{α}$$

   provided that y does not occur in M. The axiom (β) above was originally the second axiom; hence its name. We prefer our version of the theory in which the identifications are made on a syntactic level. These identifications are done in our mind and not on paper.

2. Even if initially terms are written according to the variable convention, α-conversion (or its alternative) is necessary when rewriting terms. Consider e.g. $\omega \equiv \lambda x.xx$ and $1 \equiv \lambda yz.yz$. Then

$$\begin{array}{rcl} \omega 1 & \equiv & (\lambda x.xx)(\lambda yz.yz) \\ & = & (\lambda yz.yz)(\lambda yz.yz) \\ & = & \lambda z.(\lambda yz.yz)z \\ & \equiv & \lambda z.(\lambda yz'.yz')z \end{array}$$

$$\begin{aligned} &= \lambda z z'.z z' \\ &\equiv \lambda y z.y z \\ &\equiv 1. \end{aligned}$$

3. 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 N.G. de Bruijn, see Barendregt (1984), Appendix C. In this notation $\lambda x(\lambda y.xy)$ is denoted by $\lambda(\lambda 21)$, the 2 denoting a variable bound 'two lambdas above'.

The following result provides one way to represent recursion in the λ-calculus.

Theorem 2.1.9 (Fixed point theorem).

1. $\forall F \exists X FX = X$.
   (This means that for all $F{\in}\Lambda$ there is an $X{\in}\Lambda$ such that $\lambda \vdash FX = X$.)
2. *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. 1. Define $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then
$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$.

2. By the proof of (1). Note that
$\mathsf{Y}F = (\lambda x.F(xx))(\lambda x.F(xx)) \equiv X$. ■

■

Corollary 2.1.10. *Given a term $C \equiv C[f,x]$ possibly containing the displayed free variables, then*

$$\exists F \forall X\ FX = C[F,X].$$

Here $C[F,X]$ is of course the substitution result $C[f := F][x := X]$.

Proof. Indeed, we can construct F by supposing it has the required property and calculating back:

$$\begin{array}{rrcl} & \forall X\ FX & = & C[F,X] \\ \Leftarrow & Fx & = & C[F,x] \\ \Leftarrow & F & = & \lambda x.C[F,x] \\ \Leftarrow & F & = & (\lambda f x.C[f,x])F \\ \Leftarrow & F & \equiv & \mathsf{Y}(\lambda f x.C[f,x]). \blacksquare \end{array}$$

■

This also holds for more arguments: $\exists F \forall \vec{x}\ F\vec{x} = C[F,\vec{x}]$.

As an application, terms F and G can be constructed such that for all terms X and Y

$$\begin{aligned} FX &= XF, \\ GXY &= YG(YXG). \end{aligned}$$

2.2 Lambda definability

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

Definition 2.2.1.

1. $F^n(M)$ with $n \in \mathbb{N}$ (the set of natural numbers) and $F, M \in \Lambda$, is defined inductively as follows:

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

2. The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

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

Proposition 2.2.2 (J. B. Rosser). *Define*

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

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

1. $\mathsf{A}_+\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{n+m}$.

2. $\mathsf{A}_*\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{n.m}$.

3. $\mathsf{A}_{exp}\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{(n^m)}$, *except for* $m = 0$ *(Rosser starts at 1).*

Proof. We need the following lemma.

As an application, terms F and G can be constructed such that for all terms X and Y

$$\begin{aligned} FX &= XF, \\ GXY &= YG(YXG). \end{aligned}$$

2.2 Lambda definability

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

Definition 2.2.1.

1. $F^n(M)$ with $n \in \mathbb{N}$ (the set of natural numbers) and $F, M \in \Lambda$, is defined inductively as follows:

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

2. The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

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

Proposition 2.2.2 (J. B. Rosser). *Define*

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

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

1. $\mathsf{A}_+\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{n+m}$.
2. $\mathsf{A}_*\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{n.m}$.
3. $\mathsf{A}_{exp}\mathbf{c}_n\mathbf{c}_m = \mathbf{c}_{(n^m)}$, *except for $m = 0$ (Rosser starts at 1).*

Proof. We need the following lemma.

Lemma 2.2.3.

1. $(\mathbf{c}_n x)^m(y) = x^{n*m}(y)$;
2. $(\mathbf{c}_n)^m(x) = \mathbf{c}_{(n^m)}(x)$, *for* $m > 0$.

Proof. 1. By induction on m. If $m = 0$, then LHS $= y =$ RHS. Assume (1) is correct for m (Induction Hypothesis: IH). Then

$$\begin{array}{rcl} (\mathbf{c}_n x)^{m+1}(y) & = & \mathbf{c}_n x((\mathbf{c}_n x)^m(y)) \\ & =_{IH} & \mathbf{c}_n x(x^{n*m}(y)) \\ & = & x^n(x^{n*m}(y)) \\ & \equiv & x^{n+n*m}(y) \\ & \equiv & x^{n*(m+1)}(y). \end{array}$$

2. By induction on $m > 0$. If $m = 1$, then LHS $\equiv \mathbf{c}_n x \equiv$ RHS. If (2) is correct for m, then

$$\begin{array}{rcl} \mathbf{c}_n^{m+1}(x) & = & \mathbf{c}_n(\mathbf{c}_n^m(x)) \\ & =_{IH} & \mathbf{c}_n(\mathbf{c}_{(n^m)}(x)) \\ & = & \lambda y.(\mathbf{c}_{(n^m)}(x))^n(y) \\ & =_{(1)} & \lambda y.x^{n^m * n}(y) \\ & = & \mathbf{c}_{(n^{m+1})}x. \blacksquare \end{array}$$

$\blacksquare$

Now the proof of the proposition.

1. By induction on m.
2. Use the lemma (1).
3. By the lemma (2) we have for $m > 0$

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

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

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

$\blacksquare$

We have seen that the functions plus, times and exponentiation on $\mathbb{N}$ can be represented in the λ-calculus using Church's numerals. We will show that all computable (recursive) functions can be represented.

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

Definition 2.2.4 (Booleans, conditional).

1. $\mathbf{true} \equiv \lambda xy.x$, $\mathbf{false} \equiv \lambda xy.y$.
2. If B is a Boolean, i.e. a term that is either **true**, or **false**, then
$$\mathbf{if}\ B\ \mathbf{then}\ P\ \mathbf{else}\ Q$$
can be represented by BPQ. Indeed, $\mathbf{true}PQ = P$ and $\mathbf{false}PQ = Q$.

Definition 2.2.5 (Pairing). For $M, N \in \Lambda$ write
$$[M, N] \equiv \lambda z.zMN.$$
Then
$$[M,N]\,\mathbf{true} = M$$
$$[M,N]\,\mathbf{false} = N$$
and hence $[M,N]$ can serve as an ordered pair.

Definition 2.2.6.

1. A *numeric function* is a map $f : \mathbb{N}^p \rightarrow \mathbb{N}$ for some p.
2. A numeric function f with p arguments is called λ-*definable* if one has for some combinator F
$$F\mathbf{c}_{n_1} \dots \mathbf{c}_{n_p} = \mathbf{c}_{f(n_1,\dots,n_p)} \tag{1}$$
for all $n_1, \dots, n_p \in \mathbb{N}$. If (1) holds, then f is said to be λ-*defined* by F.

Definition 2.2.7.

1. The *initial functions* are the numeric functions U^i_r, S^+, Z defined by:
$$\begin{array}{rcl} U^i_r(x_1,\dots,x_r) & = & x_i, \quad 1 \le i \le r; \\ S^+(n) & = & n+1; \\ Z(n) & = & 0. \end{array}$$
2. 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.

As we know from Chapter 2 in this handbook, 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 by a corresponding induction argument. The result is originally due to Kleene (1936).

Lemma 2.2.8. *The initial functions are λ-definable.*

Proof. Take as defining terms

$$\begin{array}{lcl} \mathsf{U}^i_p & \equiv & \lambda x_1 \cdots x_p . x_i; \\ \mathsf{S}^+ & \equiv & \lambda xyz.y(xyz) \quad (= \mathsf{A}_+ \mathbf{c}_1); \\ \mathsf{Z} & \equiv & \lambda x.\mathbf{c}_0. \blacksquare \end{array}$$

∎

Lemma 2.2.9. *The λ-definable functions are closed under composition.*

Proof. Let $g, h_1, \ldots, h_m$ be λ-defined by $G, H_1, \ldots, H_m$ respectively. Then

$$f(\vec{n}) = g(h_1(\vec{n}), \ldots, h_m(\vec{n}))$$

is λ-defined by

$$F \equiv \lambda \vec{x}.G(H_1\vec{x}) \ldots (H_m\vec{x}). \blacksquare$$

∎

Lemma 2.2.10. *The λ-definable functions are closed under primitive recursion.*

Proof. Let f be defined by

$$\begin{array}{l} f(0, \vec{n}) = g(\vec{n}) \\ f(k+1, \vec{n}) = h(f(k, \vec{n}), k, \vec{n}) \end{array}$$

where g, h are λ-defined by G, H respectively. We have to show that f is λ-definable. For notational simplicity we assume that there are no parameters $\vec{n}$ (hence $G = \mathbf{c}_{f(0)}$.) The proof for general $\vec{n}$ is similar.

If k is not an argument of h, then we have the scheme of iteration. Iteration can be represented easily in the λ-calculus, because the Church numerals are iterators. The construction of the representation of f is done in two steps. First primitive recursion is reduced to iteration using ordered pairs; then iteration is represented. Here are the details. Consider

$$T \equiv \lambda p.[\mathsf{S}^+(p\,\mathbf{true}), H(p\,\mathbf{false})(p\,\mathbf{true})].$$

Then for all k one has

$$\begin{aligned} T([\mathbf{c}_k, \mathbf{c}_{f(k)}]) &= [\mathbf{f}\mathsf{S}^+\mathbf{c}_k, H\mathbf{c}_{f(k)}\mathbf{c}_k] \\ &= [\mathbf{c}_{k+1}, \mathbf{c}_{f(k+1)}]. \end{aligned}$$

By induction on k it follows that

$$[\mathbf{c}_k, \mathbf{c}_{f(k)}] = T^k[\mathbf{c}_0, \mathbf{c}_{f(0)}].$$

Therefore

$$\mathbf{c}_{f(k)} = \mathbf{c}_k T[\mathbf{c}_0, \mathbf{c}_{f(0)}]\ \mathbf{false},$$

and f can be λ-defined by

$$F \equiv \lambda k.kT[\mathbf{c}_0, G]\ \mathbf{false}. \blacksquare$$

$\blacksquare$

Lemma 2.2.11. *The λ-definable functions are closed under minimalization.*

Proof. Let f be defined by $f(\vec{n}) = \mu m[g(\vec{n}, m) = 0]$, where $\vec{n} = n_1, \ldots, n_k$ and g is λ-defined by G. We have to show that f is λ-definable. Define

$$\mathbf{zero} \equiv \lambda n.n(\mathbf{true\ false})\mathbf{true}.$$

Then

$$\begin{aligned} \mathbf{zero}\ \mathbf{c}_0 &= \mathbf{true}, \\ \mathbf{zero}\ \mathbf{c}_{n+1} &= \mathbf{false}. \end{aligned}$$

By Corollary 2.1.10 there is a term H such that

$$H\vec{n}y = \mathbf{if}\ (\mathbf{zero}(G\vec{n}y))\ \mathbf{then}\ y\ \mathbf{else}\ H\vec{n}(\mathsf{S}^+y).$$

Set $F = \lambda\vec{n}.H\vec{x}\mathbf{c}0$. Then F λ-defines f:

$$\begin{aligned} F\mathbf{c}_{\vec{x}} &= H\mathbf{c}_{\vec{n}}\mathbf{c}_0 & & \\ &= \mathbf{c}_0, & & \text{if } G\mathbf{c}_{\vec{n}}\mathbf{c}_0 = \mathbf{c}_0, \\ &= H\mathbf{c}_{\vec{n}}\mathbf{c}_1 & & \text{else;} \\ &= \mathbf{c}_1, & & \text{if } G\mathbf{c}_{\vec{n}}\mathbf{c}_1 = \mathbf{c}_0, \\ &= H\mathbf{c}_{\vec{n}}\mathbf{c}_2 & & \text{else;} \\ &= \mathbf{c}_2, & & \text{if } \ldots \\ &= \ldots & & \end{aligned}$$

Here $\mathbf{c}_{\vec{n}}$ stands for $\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k}$. $\blacksquare$ $\blacksquare$

Theorem 2.2.12. *All recursive functions are λ-definable.*

Proof. By 2.2.8-2.2.11. ■ ■

The converse also holds. The idea is that if a function is λ-definable, then its graph is recursively enumerable because equations derivable in the λ-calculus can be enumerated. It then follows that the function is recursive. So for numeric functions we have f is recursive iff f is λ-definable. Moreover also for partial functions a notion of λ-definability exists and one has ψ is partial recursive iff ψ is λ-definable. The notions λ-definable and recursive both are intended to be formalizations of the intuitive concept of computability. Another formalization was proposed by Turing in the form of Turing computable. The equivalence of the notions recursive, λ-definable and Turing computable (for the latter see besides the original Turing, 1937, e.g., Davis 1958) Davis provides some evidence for the Church–Turing thesis that states that 'recursive' is the proper formalization of the intuitive notion 'computable'.

We end this subsection with some undecidability results. First we need the coding of λ-terms. Remember that the collection of variables is $\{v, v', v'', \ldots\}$.

Definition 2.2.13.

1. Notation. $v^{(0)} = v$; $v^{(n+1)} = v^{(n)\prime}$.
2. Let $\langle\ ,\ \rangle$ be a recursive coding of pairs of natural numbers as a natural number. Define
$$\begin{array}{rcl} \sharp(v^{(n)}) & = & \langle 0, n\rangle; \\ \sharp(MN) & = & \langle 2, \langle \sharp(M), \sharp(N)\rangle\rangle; \\ \sharp(\lambda x.M) & = & \langle 3, \langle \sharp(x), \sharp(M)\rangle\rangle. \end{array}$$
3. Notation
$$\ulcorner M \urcorner = \mathbf{c}_{\sharp M}.$$

Definition 2.2.14. Let $\mathcal{A} \subseteq \Lambda$.

1. *A is closed under* $=$ if
$$M \in \mathcal{A},\ \lambda \vdash M = N \quad \Rightarrow \quad N \in \mathcal{A}.$$
2. *A is non-trivial* if $\mathcal{A} \neq \emptyset$ and $\mathcal{A} \neq \Lambda$.
3. *A is recursive* if $\sharp\mathcal{A} = \{\sharp M \mid M \in \mathcal{A}\}$ is recursive.

The following result due to Scott is quite useful for proving undecidability results.

Theorem 2.2.15. *Let $\mathcal{A} \subseteq \Lambda$ be non-trivial and closed under $=$. Then $\mathcal{A}$ is not recursive.*

Proof. (J. Terlouw) Define

$$\mathcal{B} = \{M \mid M\ulcorner M \urcorner \in \mathcal{A}\}.$$

Suppose $\mathcal{A}$ is recursive; then by the effectiveness of the coding also $\mathcal{B}$ is recursive (indeed, $n \in \sharp\mathcal{B} \Leftrightarrow \langle 2, \langle n, \sharp\mathbf{c}_n\rangle\rangle \in \sharp\mathcal{A}$). It follows that there is an $F \in \Lambda^0$ with

$$\begin{aligned} M \in \mathcal{B} &\Leftrightarrow F\ulcorner M \urcorner = \mathbf{c}_0; \\ M \notin \mathcal{B} &\Leftrightarrow F\ulcorner M \urcorner = \mathbf{c}_1. \end{aligned}$$

Let $M_0 \in \mathcal{A}, M_1 \notin \mathcal{A}$. We can find a $G \in \Lambda$ such that

$$\begin{aligned} M \in \mathcal{B} &\Leftrightarrow G\ulcorner M \urcorner = M_1 \notin \mathcal{A}, \\ M \notin \mathcal{B} &\Leftrightarrow G\ulcorner M \urcorner = M_0 \in \mathcal{A}. \end{aligned}$$

[Take $Gx = \mathbf{if}\ \mathbf{zero}(Fx)\ \mathbf{then}\ M_1\ \mathbf{else}\ M_0$, with **zero** defined in the proof of 2.2.11.] In particular

$$\begin{aligned} G \in \mathcal{B} &\Leftrightarrow G\ulcorner G \urcorner \notin \mathcal{A} &\Leftrightarrow_{\text{Def}} G \notin \mathcal{B}, \\ G \notin \mathcal{B} &\Leftrightarrow G\ulcorner G \urcorner \in \mathcal{A} &\Leftrightarrow_{\text{Def}} G \in \mathcal{B}, \end{aligned}$$

a contradiction. ■ █

The following application shows that the lambda calculus is not a decidable theory.

Corollary 2.2.16 (Church). *The set*

$$\{M \mid M = \mathbf{true}\}$$

is not recursive.

Proof. Note that the set is closed under $=$ and is nontrivial. ■ █

2.3 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'.

5. Self-reflection

We present the following fact with a proof depending on another fact.

5.1. Fact. Let x, y be to distinct variables (e.g. x, x' or x'', x'''). Then

$$x \neq_\lambda y.$$

Proof. Use Fact 4.17. If $x =_l y$, then x has two normal forms: itself and y. ■

5.2. Application. $\mathsf{K} \neq_\lambda \mathsf{I}$.

Proof. Suppose $\mathsf{K} = \mathsf{I}$. Then

$$\begin{array}{rcl} x & = & \mathsf{K}xy \\ & = & \mathsf{IK}xy \\ & = & \mathsf{KI}xy \\ & = & \mathsf{I}y \\ & = & y, \end{array}$$

a contradiction. ■

5.3. Application. There is no term P_1 such that $P_1(xy) =_\lambda x$.

Proof. If P_1 would exist, then as $\mathsf{K}xy = x = \mathsf{I}x$ one has

$$\mathsf{K}x = P_1((\mathsf{K}x)y) = P_1(\mathsf{I}x) = \mathsf{I}.$$

Therefore we obtain the contradiction

$$x = \mathsf{K}xy = \mathsf{I}y = y. \ ■$$

In a sense this is a pity. The agents, that lambda terms are, cannot separate two of them that are together. When we go over to codes of lambda terms the situation changes. The situation is similar for proteins that cannot always cut into parts another protein, but are able to have this effect on the code of the proteins, the DNA.

Data types

Before we enter the topic of coding of lambda terms, it is good to have a look at some datatypes.

Context-free languages can be considered as algebraic data types. Consider for example

$$\boxed{\begin{array}{rcl} S & \to & 0 \\ S & \to & S^+ \end{array}}$$

This generates the language

$$\text{Nat} = \{0, 0^+, 0^{++}, 0^{+++}, \ldots\}$$

that is a good way to represent the natural numbers

$$0, 1, 2, 3, \ldots$$

Another example generates binary trees.

$$\boxed{\begin{array}{lcl} S & \to & \spadesuit \\ S & \to & \bullet SS \end{array}}$$

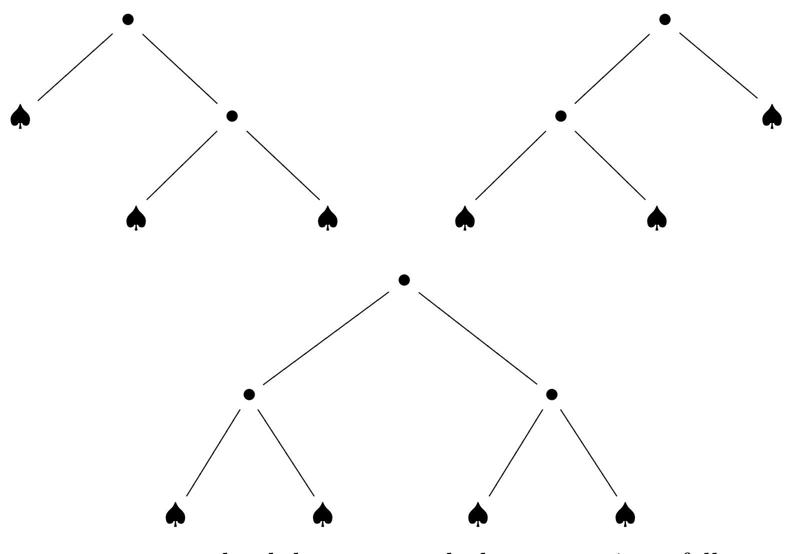
generating the language that we call *Tree*. It is not necessary to use parentheses. For example the words

$$\begin{array}{l} \bullet\spadesuit\bullet\spadesuit\spadesuit \\ \bullet\bullet\spadesuit\spadesuit\spadesuit \\ \bullet\bullet\spadesuit\spadesuit\bullet\spadesuit\spadesuit \end{array}$$

are in Tree. With parentrheses and commas these expressions become more readable for humans, but these auxiliary signs are not necessary:

$$\begin{array}{l} \bullet(\spadesuit, \bullet(\spadesuit, \spadesuit)) \\ \bullet(\bullet(\spadesuit, \spadesuit), \spadesuit) \\ \bullet(\bullet(\spadesuit, \spadesuit), \bullet(\spadesuit, \spadesuit)) \end{array}$$

These expressions have as 'parse trees' respectively the following:



One way to represent as lambda terms such data types is as follows.

5.4. Definition. (Böhm and Berarducci)
(i) An element of Nat, like 0^{++} will be represented first like

$$s(sz)$$

and then like

$$\lambda sz.s(sz).$$

If n is in Nat we write $\ulcorner n \urcorner$ for this lambda term. So $\ulcorner 2 \urcorner \equiv \lambda sz.s(sz)$, $\ulcorner 3 \urcorner \equiv \lambda sz.s(s(sz))$. Note that $\ulcorner n \urcorner \equiv \lambda sz.s^n z \equiv \mathsf{C}_n$.

(ii) An element of Tree, like $\bullet\spadesuit\bullet\spadesuit\spadesuit$ will be represented first by

$$bs(bss)$$

and then by

$$\lambda bs.bs(bss).$$

This lambda term is denoted by $\ulcorner\bullet\spadesuit\bullet\spadesuit\spadesuit\urcorner$, in this case. In general a tree t in Tree will be represented as lambda term $\ulcorner t\urcorner$.

Now it becomes possible to compute with trees. For example making the mirror image is performed by the term

$$F_{\text{mirror}} \equiv \lambda tbs.t(\lambda pq.bqp)s.$$

The operation of enting one tree at the endpoints of another tree is performed by

$$F_{\text{enting}} \equiv \lambda t_1t_2bs.t_1b(t_2bs).$$

The attentive reader is advised to make exercises 5.4 and 5.5.

Tuples and projections

For the efficient coding of lambda terms as lambda terms a different representation of datatypes is needed. First we find a way to connect terms together in such a way, that the components can be retrieved easily.

5.5. Definition. Let $\vec{M} \equiv M_1, \ldots, M_n$ be a sequence of lambda terms. Define

$$\langle M_1, \ldots, M_n\rangle \equiv \lambda z.zM_1, \ldots, M_n.$$

Here the variable z should not be in any of the M's. Define

$$U_i^n \equiv \lambda x_1, \ldots, x_n.x_i.$$

5.6. Proposition. *For all natural numbers i, n with $1 \le i \le n$, one has*

$$\langle M_1, \ldots, M_n\rangle U_i^n = M_i.$$

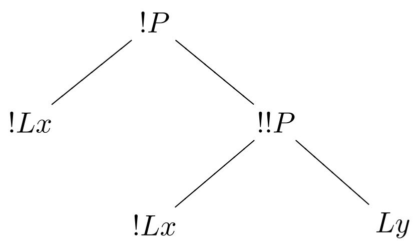
5.7. Corollary. *Define $P_i^n \equiv \lambda z.zU_i^n$. Then $P_i^n\langle M_1, \ldots, M_n\rangle = M_i$.*

Now we introduce a new kind of binary trees. At the endpoints there is not a symbol $\spadesuit$ but a variable x made into a leaf Lx. Moreover, any such tree may get ornamented with a !. Such binary trees we call labelled trees or simply ltrees.

5.8. Definition. The data type of ltrees is defined by the following context-free grammar. The start symbol is `ltree`.

`ltree`	$\rightarrow$	L `var`
`ltree`	$\rightarrow$	P `ltree` `ltree`
`ltree`	$\rightarrow$	! `ltree`
`var`	$\rightarrow$	x
`var`	$\rightarrow$	`var`$'$

A typical ltree is $!P!Lx!!P!LxLy$ or more readably $!P(!Lx, !!P(!Lx, Ly))$. It can be representated as a tree as follows.



We say that for ltree there are three *constructors*. A binary constructor P that puts two trees together, and two unary constructors L and !. L makes from a variable an ltree and ! makes from an ltree another one.

Now we are going to represent these expressions as lambda terms.

5.9. Definition. (Böhm, Piperno and Guerrini)
(i) We define three lambda terms $F_L, F_P, F_!$ to be used for the representation of ltree.

$$\begin{array}{rcl} F_L & \equiv & \lambda xe.eU_1^3xe; \\ F_P & \equiv & \lambda xye.eU_2^3xye; \\ F_! & \equiv & \lambda xe.eU_3^3xe. \end{array}$$

These definitions are a bit more easy to understand if written according to their intended use (do exercise 5.7).

$$\begin{array}{rcl} F_Lx & = & \lambda e.eU_1^3xe; \\ F_Pxy & = & \lambda e.eU_2^3xye; \\ F_!x & = & \lambda e.eU_3^3xe. \end{array}$$

(ii) For an element t of ltree we define the representing lambda term $\ulcorner t \urcorner$.

$$\begin{array}{rcl} \ulcorner Lx \urcorner & = & F_Lx; \\ \ulcorner Pt_1t_2 \urcorner & = & F_P\ulcorner t_1 \urcorner\ulcorner t_2 \urcorner; \\ \ulcorner !t \urcorner & = & F_!\ulcorner t \urcorner. \end{array}$$

Actually this is just a mnemonic. We want that the representations are normal forms, do not compute any longer.

$$\begin{array}{rcl} \ulcorner Lx \urcorner & \equiv & \lambda e.eU_1^3xe; \\ \ulcorner Pt_1t_2 \urcorner & \equiv & \lambda e.eU_2^3\ulcorner t_1 \urcorner\ulcorner t_2 \urcorner e; \\ \ulcorner !t \urcorner & \equiv & \lambda e.eU_3^3\ulcorner t \urcorner e. \end{array}$$

The representation of the data was chosen in such a way that computable function on them can be easily represented. The following result states that there exist functions on the represented labelled trees such that their action on a composed tree depend on the components and that function in a given way.

5.10. PROPOSITION. *Let A_1, A_2, A_3 be given lambda terms. Then there exists a lambda term H such that*[11].

$$\begin{array}{rcl} H(F_L x) & = & A_1 x H \\ H(F_P xy) & = & A_2 xy H \\ H(F_! x) & = & A_3 x H \end{array}$$

PROOF. We try $H \equiv \langle\langle B_1, B_2, B_3\rangle\rangle$ where the $\vec{B}$ are to be determined.

$$\begin{array}{rcl} H(F_L x) & = & \langle\langle B_1, B_2, B_3\rangle\rangle(F_L x) \\ & = & F_L x\langle B_1, B_2, B_3\rangle \\ & = & \langle B_1, B_2, B_3\rangle U_1^3 x\langle B_1, B_2, B_3\rangle \\ & = & U_1^3 B_1, B_2, B_3 x\langle B_1, B_2, B_3\rangle \\ & = & B_1 x\langle B_1, B_2, B_3\rangle \\ & = & A_1 x H, \end{array}$$

provided that $B_1 \equiv \lambda xb.A_1 x\langle b\rangle$. Similarly

$$\begin{array}{rcl} H(F_P xy) & = & B_2 xy\langle B_1, B_2, B_3\rangle \\ & = & A_2 xy H, \end{array}$$

provided that $B_2 \equiv \lambda xyb.A_2 xy\langle b\rangle$. Finally,

$$\begin{array}{rcl} H(F_! x) & = & B_3 x\langle B_1, B_2, B_3\rangle \\ & = & A_3 x H, \end{array}$$

provided that $B_3 \equiv \lambda xb.A_3 x\langle b\rangle$. ∎

Stil we have as goal to represent lambda terms as lambda terms in nf, such that decoding is possible by a fixed lambda term. Moreover, finding the code of the components of a term M should be possible from the code of M, again using a lambda term. To this end the (represented) constructors of ltree, $F_L, F_P, F_!$, will be used.

5.11. DEFINITION. (Mogensen) Define for a lambda term M its code $\ulcorner M\urcorner$ as follows.

$$\begin{array}{rclcl} \ulcorner x\urcorner & \equiv & \lambda e.eU_1^3 xe & = & F_L x; \\ \ulcorner MN\urcorner & \equiv & \lambda e.eU_2^3\ulcorner M\urcorner\ulcorner N\urcorner e & = & F_P\ulcorner M\urcorner\ulcorner N\urcorner; \\ \ulcorner \lambda x.M\urcorner & \equiv & \lambda e.eU_3^3(\lambda x.\ulcorner M\urcorner)e & = & F_!(\lambda x.\ulcorner M\urcorner). \end{array}$$

[11]A weaker requirement is the following, where H of a composed ltree depends on H of the components in a given way:

$$\begin{array}{rcl} H(F_L x) & = & A_1 x(Hx) \\ H(F_P xy) & = & A_2 xy(Hx)(Hy) \\ H(F_! x) & = & A_3 x(Hx) \end{array}$$

This is called primitive recursion, whereas the proposition provides (general) recursion.

The trick here is to code the lambda with lambda itself, one may speak of an inner model of the lambda calculus in itself. Putting the ideas of Mogensen [1992] and Böhm et al. [1994] together, as done by Berarducci and Böhm [1993], one obtains a very smooth way to create the mechanism of reflection the lambda calculus. The result was already proved in Kleene [1936][12].

5.12. Theorem. *There is a lambda term* E *(evaluator or self-interpreter) such that*

$$\begin{array}{rcl} \mathsf{E}\ulcorner x\urcorner &=& x; \\ \mathsf{E}\ulcorner MN\urcorner &=& \mathsf{E}\ulcorner M\urcorner(\mathsf{E}\ulcorner N\urcorner); \\ \mathsf{E}\ulcorner \lambda x.M\urcorner &=& \lambda x.(\mathsf{E}\ulcorner M\urcorner). \end{array}$$

It follows that for all lambda terms M *one has*

$$\mathsf{E}\ulcorner M\urcorner = M.$$

Proof. By Proposition 5.10 for arbitary $A_1, \ldots, A_3$ there exists an E such that

$$\begin{array}{rcl} \mathsf{E}(F_L x) &=& A_1 x\mathsf{E}; \\ \mathsf{E}(F_P mn) &=& A_2 mn\mathsf{E}; \\ \mathsf{E}(F_{!}p) &=& A_3 p\mathsf{E}. \end{array}$$

If we take $A_1 \equiv \mathsf{K}$, $A_2 \equiv \lambda abc.ca(cb)$ and $A_3 \equiv \lambda abc.b(ac)$, then this becomes

$$\begin{array}{rcl} \mathsf{E}(F_L x) &=& x; \\ \mathsf{E}(F_P mn) &=& \mathsf{E}m(\mathsf{E}n); \\ \mathsf{E}(F_{!}p) &=& \lambda x.(\mathsf{E}(px)). \end{array}$$

But then (do exercise 5.9)

$$\begin{array}{rcl} \mathsf{E}(\ulcorner x\urcorner) &=& x; \\ \mathsf{E}(\ulcorner MN\urcorner) &=& \mathsf{E}\ulcorner M\urcorner(\mathsf{E}\ulcorner N\urcorner); \\ \mathsf{E}(\ulcorner \lambda x.M\urcorner) &=& \lambda x.(\mathsf{E}(\ulcorner M\urcorner)). \end{array}$$

That E is a self-interpreter, i.e. $\mathsf{E}\ulcorner M\urcorner = M$, now follows by induction on M. ■

5.13. Corollary. *The term* $\langle\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle\rangle$ *is a self-interpreter for the lambda calculus with the coding defined in Definition 5.11.*

Proof. $\mathsf{E} \equiv \langle\langle B_1, B_2, B_3\rangle\rangle$ with the $\vec{B}$ coming from the $A_1 \equiv \mathsf{K}$, $A_2 \equiv \lambda abc.ca(cb)$ and $A_3 \equiv \lambda abc.b(ac)$. Looking at the proof of 5.10 one sees

$$\begin{array}{rcl} B_1 &=& \lambda xz.A_1 x\langle z\rangle \\ &=& \lambda xz.x \\ &=& \mathsf{K}; \end{array}$$

[12]But only valid for lambda terms M without free variables.

$$\begin{array}{rcl}
B_2 & = & \lambda xyz.A_2xy\langle z\rangle \\
 & = & \lambda xyz.\langle z\rangle x(\langle z\rangle y) \\
 & = & \lambda xyz.xz(yz) \\
 & = & \mathsf{S}; \\
B_3 & = & \lambda xz.A_3x\langle z\rangle \\
 & = & \lambda xz.(\lambda abc.b(ac))x\langle z\rangle \\
 & = & \lambda xz.(\lambda c.xcz) \\
 & \equiv & \lambda xzc.xcz \\
 & \equiv & \lambda xyz.xzy \\
 & \equiv & \mathsf{C}, \qquad\qquad \text{see exercise 4.6.}
\end{array}$$

Hence $\mathsf{E} = \langle\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle\rangle$. ■

This term

$$\mathsf{E} = \langle\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle\rangle$$

is truly a tribute to

Kleene, Stephen Cole
(1909-1994)

(using the family-name-first convention familiar from scholastic institutions) who invented in 1936 the first self-interpreter for the lambda calculus[13].

The idea of a language that can talk about itself has been heavily used with higher programming languages. The way to translate ('compile') these into machine languages is optimized by writing the compiler in the language itself (and run it the first time by an older *ad hoc* compiler). This possibility of efficiently executed higher programming languages was first put into doubt, but was realized by mentioned reflection since the early 1950-s and other optimalizations. The box of Pandora of the world of IT was opened.

The fact that such extremely simple (compared to a protein like titin with slightly less than 27000 aminoacids) self interpreter is possible gives hope to understand the full mechanism of cell biology and evolution. In Buss and Fontana [1994] evolution is modelled using lambda terms.

Exercises

5.1. Show that

$$\begin{array}{rcl}
\mathsf{K} & \neq_\lambda & \mathsf{S}; \\
\mathsf{I} & \neq_\lambda & \mathsf{S}.
\end{array}$$

.

[13]Kleene's construction was much more involved. In order to deal with the 'binding effect' of λx lambda terms where first translated into **CL**before the final code was obtained. This causes some technicalities that make the original E more complex.

5.2. Show that there is no term P_2 such that $P_2(xy) =_\lambda y$.

5.3. Construct all elements of Tree with exactly four $\spadesuit$s in them.

5.4. Show that

$$\begin{array}{rcl}
F_{\text{mirror}}\ulcorner \bullet \spadesuit \bullet \spadesuit \spadesuit \urcorner & = & \ulcorner \bullet \bullet \spadesuit \spadesuit \spadesuit \urcorner; \\
F_{\text{mirror}}\ulcorner \bullet \bullet \spadesuit \spadesuit \spadesuit \urcorner & = & \ulcorner \bullet \spadesuit \bullet \spadesuit \spadesuit \urcorner; \\
F_{\text{mirror}}\ulcorner \bullet \bullet \spadesuit \spadesuit \bullet \spadesuit \spadesuit \urcorner & = & \ulcorner \bullet \bullet \spadesuit \spadesuit \bullet \spadesuit \spadesuit \urcorner.
\end{array}$$

5.5. Compute $F_{\text{enting}}\ulcorner \bullet \spadesuit \bullet \spadesuit \spadesuit \urcorner \ulcorner \bullet \bullet \spadesuit \spadesuit \spadesuit \urcorner$.

5.6. Define terms P_i^n such that for $1 \leq i \leq n$ one has

$$P_i^n \langle M_1, \ldots, M_n \rangle = M_i.$$

5.7. Show that the second set of three equations in definition 5.9 follows from the first set.

5.8. Show that given terms A_1, A_2, A_3 there exists a term H such that the scheme of primitive recurion, see footnote 5.10 is valid.

5.9. Show the last three equations in the proof of Theorem 5.12.

5.10. Construct lambda terms P_1 and P_2 such that for all terms M, N

$$P_1 \ulcorner MN \urcorner = M \ \& \ P_2 \ulcorner MN \urcorner = N.$$

References

Alberts, B. et al. [1997]. *The Cell*, Garland.

Barendregt, H. P. [1984]. *The Lambda Calculus, its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics 103, revised edition, North-Holland Publishing Co., Amsterdam.

Barendregt, H. P. [1997]. The impact of the lambda calculus in logic and computer science, *Bull. Symbolic Logic* **3**(2), pp. 181–215.

Berarducci, Alessandro and Corrado Böhm [1993]. A self-interpreter of lambda calculus having a normal form, *Computer science logic (San Miniato, 1992)*, Lecture Notes in Comput. Sci. 702, Springer, Berlin, pp. 85–99.

Blackmore, S. [2004]. *Consciousness, an Introduction*, Oxford University Press, Oxford.

Böhm, Corrado, Adolfo Piperno and Stefano Guerrini [1994]. λ-definition of function(al)s by normal forms, *Programming languages and systems—ESOP '94 (Edinburgh, 1994)*, Lecture Notes in Comput. Sci. 788, Springer, Berlin, pp. 135–149.

Buss, L.W. and W. Fontana [1994]. 'the arrival of the fittest': Toward a theory of biological organization, *Bulletin of Mathematical Biology* **56**(1), pp. 1–64.

Chalmers, D. [1996]. *The Conscious Mind, Towards a Fundamental Theory*, Oxford University Press, Oxford.

Chomsky, N. [1956]. Three models of the description of language, *IRE Transactions on Information Theory* **2**(3), pp. 113–124.

Church, A. [1932]. A set of postulates for the foundation of logic, *Annals of Mathematics, second series* **33**, pp. 346–366.

Church, A. [1936]. An unsolvable problem of elementary number theory, *American Journal of Mathematics* **58**, pp. 345–363.

Curry, H. B. [1930]. Grundlagen der kombinatorischen Logic,, *American Journal of Mathematics* **52**, pp. 509–536, 789–834.

Dennet, D. [1993]. *Consciousness Explained*, Penguin Books.

Goldstein, J. [1983]. *The Experience of Insight*, Shambhala.

Hofstadter, D. [1979]. *Gödel Escher Bach, An Eternal Golden Braid*, Harvester Press.

Howe, D. [1992]. Reflecting the semantics of reflected proof, *Proof Theory, ed. P. Aczel*, Cambridge University Press, pp. 229–250.

Kleene, S. C. [1936]. Lambda-definability and recursiveness, *Duke Mathematical Journal* **2**, pp. 340–353.

Kozen, Dexter C. [1997]. *Automata and computability*, Undergraduate Texts in Computer Science, Springer-Verlag, New York.

Menninger, K., M. Mayman and P. Pruyser [1963]. *The Vital Balance. The Life Process in Mental Health and Illness*, Viking.

Mogensen, Torben Æ. [1992]. Efficient self-interpretation in lambda calculus, *J. Funct. Programming* **2**(3), pp. 345–363.

Peitsch, M.C., D.R. Stampf, T.N.C. Wells and J.L. Sussman [1995]. The swiss-3dimage collection and pdb-browser on the world-wide web, *Trends in Biochemical Sciences* **20**, pp. 82–84. URL: `<www.expasy.org>`.

Schönfinkel, M. [1924]. Über die Bausteine der mathematischen Logik, *Mathematische Annalen* **92**, pp. 305–316.

Smullyan, R. [1992]. *Gödel's Incompleteness Theorems*, Oxford University Press.

Stapp, H. [1996]. The hard problem: A quantum approach, *Journal of Consciousness Studies* **3**(3), pp. 194–210.

Tarski, A. [1933/1995]. *Introduction to Logic*, Dover.

Turing, A.M. [1936]. On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society, Series 2* **42**, pp. 230–265.

Yates, M. [1998]. What computers can't do, *Plus* **5**.
URL: `<plus.maths.org/issue5/index.html>`.

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

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

(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.

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

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

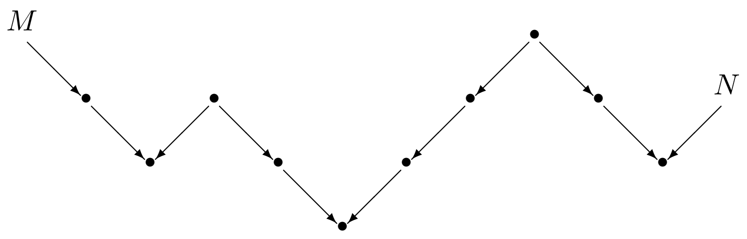
4.3. Example. (i) Define

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

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

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

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



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

$$\begin{tikzcd} \mathsf{\mathbf{KI}}\boldsymbol{\Omega} \arrow[dr] & & & \\ & (\lambda y.\mathsf{\mathbf{I}})\boldsymbol{\Omega} \arrow[dr] & & \mathsf{\mathbf{II}} \arrow[dl] \\ & & \mathsf{\mathbf{I}} & \end{tikzcd}$$

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 $\to_\beta$. Then the result follows by transitivity. $\square$

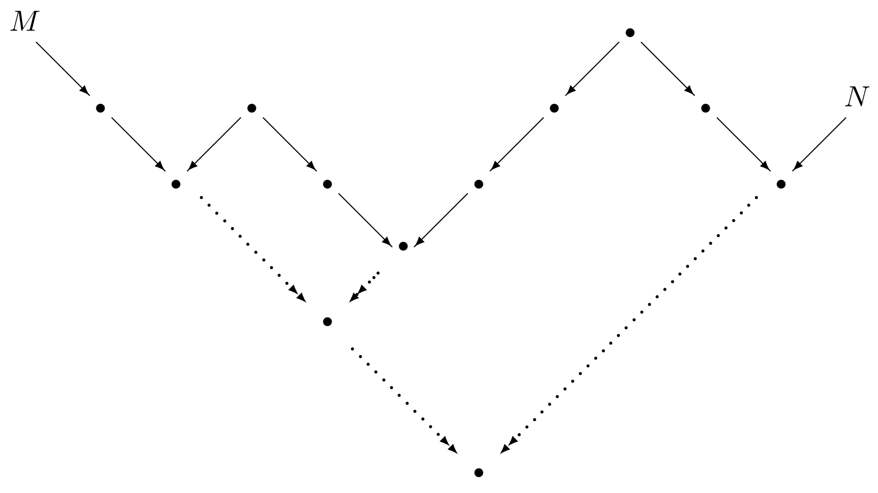
4.9. Church-Rosser Theorem. *If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$; in diagram*

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
N_1 \arrow[dr, dotted] & & N_2 \arrow[dl, dotted] \\
 & N_3 &
\end{tikzcd}$$

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

$$\begin{tikzcd}
M \arrow[dr] \arrow[rr, phantom, "\text{(IH)}"] & & N' \arrow[dl] \arrow[dr] \arrow[rr, phantom, "\text{(IH)}"] & & N \arrow[dl] \\
& L_1 \arrow[dr, dotted] \arrow[rr, phantom, "\text{(CR)}"] & & L_2 \arrow[dl, dotted] & \\
& & L & &
\end{tikzcd}$$

□

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. $\boldsymbol{\lambda} \nvdash$ **true** $=$ **false**. Otherwise **true** $=_\beta$ **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:

$$\begin{tikzcd}
& M \arrow[dl, "\beta"'] \arrow[ddr, "\beta"] & \\
N_1 \arrow[ddr, dotted, "\beta"'] & & \\
& & N_2 \arrow[dl, dotted, "\beta"] \\
& N_3 &
\end{tikzcd}$$

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.

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

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

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

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.

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

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.

$$\begin{tikzcd} M' \arrow[r, dotted, two heads, "\underline{\beta}"'] \arrow[d, "|\ |"'] & N' \arrow[d, "|\ |"] \\ M \arrow[r, two heads, "\beta"'] & N \end{tikzcd} \qquad \begin{array}{l} M', N' \in \underline{\Lambda}, \\ M, N \in \Lambda. \end{array}$$

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)

$$\begin{tikzcd}
M \arrow[rr, two heads, "\underline{\beta}"'] \arrow[d, "\varphi"'] & & N \arrow[d, "\varphi"] \\
\varphi(M) \arrow[rr, dotted, two heads, "\beta"'] & & \varphi(N)
\end{tikzcd} \qquad M, N \in \underline{\Lambda}.$$

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.

$$\begin{tikzcd}
& M \arrow[dl, "|\ |"'] \arrow[dr, "\varphi"] & \\
N \arrow[rr, dotted, two heads, "\beta"'] & & L
\end{tikzcd} \qquad \begin{array}{l} M \in \underline{\Lambda}, \\ N, L \in \Lambda. \end{array}$$

PROOF. By induction on the structure of M. □

4.18. STRIP LEMMA.

$$\begin{tikzcd}
& M \arrow[dl, "\beta"'] \arrow[ddrr, "\beta"] & & \\
N_1 \arrow[ddr, dotted, two heads, "\beta"'] & & & \\
& & & N_2 \arrow[dll, dotted, two heads, "\beta"] \\
& N_3 & &
\end{tikzcd} \qquad M, N_1, N_2, N_3 \in \Lambda.$$

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

$$\begin{tikzcd}
 & M \arrow[dl, "\beta"'] \arrow[rrrdd, "\beta"] & & & & \\
N_1 \arrow[rrrdd, dotted, "\beta"'] & & M' \arrow[ul, "| \, |"'] \arrow[ll, "\varphi"] \arrow[rrrdd, dotted, "\beta"] & & & \\
 & & & & N_2 \arrow[dl, dotted, "\beta"'] & \\
 & & & N_3 & & N_2' \arrow[ul, "| \, |"'] \arrow[ll, "\varphi"]
\end{tikzcd}$$

which proves the Strip Lemma. □

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

$$\begin{tikzcd}
 & & & & M \arrow[dl] \arrow[dr] & \\
 & & & M_1 \arrow[dl] \arrow[dr, dotted] & & N_2 \arrow[dl, dotted] \\
 & & \bullet \arrow[dl, dotted, no head] \arrow[dr, dotted] & & \bullet \arrow[dl, dotted] & \\
 & \bullet \arrow[dl] & & \bullet \arrow[dl, dotted, no head] & & \\
N_1 \arrow[dr, dotted] & & \bullet \arrow[dl, dotted] & & & \\
 & \bullet & & & &
\end{tikzcd}$$

□

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 $\rightarrow_\beta$.

4.21. Example. $G_\beta(\mathbf{I}(\mathbf{I}x))$ is

$$\begin{tikzcd}
\mathbf{I}(\mathbf{I}x) \arrow[d, bend left] \arrow[d, bend right] \\
\mathbf{I}x \arrow[d] \\
x
\end{tikzcd}$$

sometimes simply drawn as

$$\begin{tikzcd}
\bullet \arrow[d, bend left] \arrow[d, bend right] \\
\bullet \arrow[d] \\
\bullet
\end{tikzcd}$$

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*

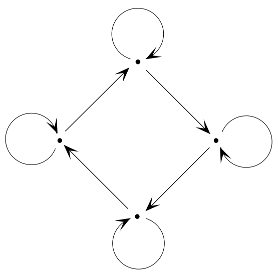
$$\begin{aligned} X_1 &\twoheadrightarrow F_1X_1 \cdots X_n, \\ &\vdots \\ X_n &\twoheadrightarrow F_nX_1 \cdots X_n. \end{aligned}$$

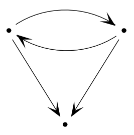
Proof. As for the equational Multiple Fixedpoint Theorem 3.17, but now using $\mathbf{\Theta}$. $\square$

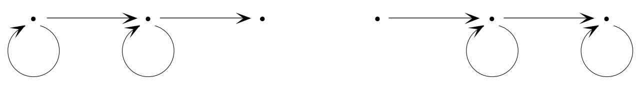
Exercises

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

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





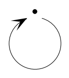


4.3. Show that there is no $F \in \Lambda$ such that for all $M, N \in \Lambda$

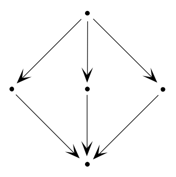
$$F(MN) = M.$$

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

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

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) $\mathbf{I}\mathbf{I}(\mathbf{I}\mathbf{I}\mathbf{I})$.

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.]