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 $F\,A$) 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 $F\,F$, 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

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

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 $V = \{v, v', v'', \ldots\}$ 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. $x, y, z, \ldots$ denote arbitrary variables;
   $M, N, L, \ldots$ denote arbitrary λ-terms.

2. As already mentioned informally, the following abbreviations are used:

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

and

$$
\lambda x_1 \cdots x_n.M \text{ stands for } (\lambda x_1(\lambda x_2(\ldots(\lambda x_n(M))..))).
$$

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

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

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

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

$$
\begin{aligned}
M &= M; \\
M = N &\Rightarrow N = M; \\
M = N, N = L &\Rightarrow M = L; \\
M = M' &\Rightarrow MZ = M'Z; \\
M = M' &\Rightarrow ZM = ZM'; \\
M = M' &\Rightarrow \lambda x.M = \lambda x.M'. \qquad (\xi)
\end{aligned}
$$

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*.

$$\lambda x.M = \lambda y.M[x := y], \qquad (\alpha)$$

   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

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

$$= \quad \lambda zz'.zz'$$
$$\equiv \quad \lambda yz.yz$$
$$\equiv \quad 1.$$

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
$$\mathsf{Y}F = (\lambda x.F(xx))(\lambda x.F(xx)) \equiv X. \blacksquare$$

$\blacksquare$

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:

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

$\blacksquare$

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.