- reduction property:
  if an expression has a meaning, then all subexpressions have a meaning. If any subexpression is replaced by its meaning, then the meaning of the expression remains unchanged.

5.3 The λ-Calculus

The λ-calculus, first published by Alonzo Church [Chur32, ChRo36, Ros84], is the most general reduction calculus which formalizes the concept of evaluating abstract algorithms by program transformations.[9] It is primarily a theory of computable functions which deals with elementary properties of operators, with applications of operators to operands, and with the systematic construction of complex operators (functions/algorithms) from more primitive ones.

In fact, the λ-calculus is based on little more than a well-defined concept of substitution of variables by expressions with strictly functional properties. However, this concept suffices to develop a precise understanding of the problems and issues discussed in section 5.1. Moreover, it permits a precise definition of the semantics of abstract algorithms based on the equivalence of λ-expressions, and of the notion of computable functions.

The λ-calculus may be considered the paradigm for all functional programming languages and, with restrictions, for many imperative programming languages as well [Lan65, Lan66a, Lan66b].

We will here introduce the λ-calculus only as far as is necessary to understand the machine concepts and implementations discussed in the remainder of the book. For a more comprehensive study the reader is referred to [CuFe58, CuHiSe72, Bar81, HiSe86], another brief introduction may be found in [Weg71].

The term λ-calculus derives from the notation that is being used to represent functions. This notation appears to be rather unusual when compared to (recursive) function equations. However, as the λ-calculus does not really distinguish between operators and operands other than by syntactical positions within an expression, the λ-notation greatly facilitates the representation of a function as an object of either kind. Moreover, the λ-notation allows a precise definition of the binding status of variable occurrences.

The representation of functions in the λ-calculus is almost identical to the one we introduced in chapter 4.

Let

$$f(x_1, \ldots, x_n) = expr$$

[9] The contents of this section are in part based on notes privately communicated by W. Lippe and F. Simon [LiSi86].

define f as a non-recursive function of n variables $x_1, \ldots, x_n$. In the λ-calculus, f is equivalently denoted as

$$f = \lambda x_1 \ldots x_n.expr\ .$$

The right-hand side of this equation is said to be an abstraction (of the variables $x_1, \ldots, x_n$ from $expr$): $\lambda x_1 \ldots x_n$ specifies a binder for free occurrences of the variables $x_1, \ldots, x_n$ in the function body expression $expr$.

An application

$$f(arg\_1, \ldots, arg\_r)$$

assumes in the λ-notation the form

$$(f\ arg\_1 \ldots arg\_r) = (\lambda x_1 \ldots x_n.expr\ arg\_1 \ldots arg\_r)$$

(where r must not necessarily be equal to n). In particular, we have

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

In order to keep the formal apparatus simple and concise, the λ-calculus considers, without loss of generality, functions of one variable only. This is possible due to a discovery by Curry and Schoenfinkel [Scho24, Curr29], according to which functions of n variables may be equivalently represented as n-fold nestings of functions of one variable (i.e., of unary functions):

$$f = \lambda x_1 \ldots x_n.expr \equiv \lambda x_1.\lambda x_2.\ \ldots \lambda x_n.expr\ .$$

Using this so-called curried notation, we may specify the application of f to one argument, say arg_1, as

$$(f\ arg\_1) = (\lambda x_1.\ \ldots \lambda x_n.\ expr\ arg\_1)\ ,$$

and its application to r arguments $arg\_1, \ldots,\ arg\_r$ as an r-fold nesting of applications of unary functions:

$$(f\ arg\_1 \ldots arg\_r) \equiv (\ldots((f\ arg\_1)\ arg\_2) \ldots arg\_r)\ .$$ [10]

When using only unary functions, we have the following syntax for the construction of well-formed formulas of the λ-calculus, also referred to as λ-expressions or λ-terms: let V be a finite set of variables and C be a finite set of constants other than variables, i.e., $V \cap C = \emptyset$. Then

[10] In an equivalent constructor syntax notation the parentheses surrounding an application would have to be replaced by an applicator preceding it:

$$\mathbf{ap}^{(r+1)} f\ arg\_1 \ldots arg\_r \equiv \mathbf{ap}^{(2)} \ldots \mathbf{ap}^{(2)}\ \mathbf{ap}^{(2)}\ f\ arg\_1\ arg\_2 \ldots arg\_r\ .$$

- all variables $x \in V$ and constants $c \in C$ are atomic λ-terms;
- $(M\ N)$ is a λ-term called application, if M and N are λ-terms;
- $\lambda x.M$ is a λ-term called abstraction, if $x \in V$ is a variable and M is a λ-term.[11]

We use

- capital letters to denote λ-terms,
- the lower case letters u, v, w, x, y, z to denote variables,
- all other lower case letters, character strings, numbers and special (function) symbols to denote constants,
- E_λ to denote the set of legitimate λ-terms.

The application $(M\ N)$ represents the result of applying the λ-term M to the λ-term N. M and N are respectively called operator and operand (or rator and rand for short). Alternatively, we may consider M and N to be a function and an argument, respectively, although this interpretation is somewhat misleading.

The λ-term $\lambda x.M$ definitely constitutes a function (of the variable x). Its value for some argument N is computed by the substitution of all free occurrences of x in M by N. This can, however, only be accomplished by an application of the form

$$(\lambda x.M\ N)$$

in which $\lambda x.M$ is in operator position and N is in operand position.

Consider, as an example, the application

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

which transforms, in two steps, into $\lambda y.y$.

If the λ-terms within the application are interchanged, we obtain

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

and, after one transformation step, $\lambda x.(x\ x)$.

[11] Note that in some of the standard literature on the λ-calculus abstractions are also denoted as $(\lambda x\ M)$.

In either case, we apply a function to a function and reproduce, as a result, the function in operand position.[12]

Parentheses in λ-terms may be omitted as applications are, in compliance with the curried notation, assumed to be left-associative.

Literal or syntactical equality of two λ-terms M and N will be denoted as

$$M =_S N.$$

It is assumed that

$$((M\ N) =_S (P\ Q)) \Rightarrow ((M =_S P) \wedge (N =_S Q))$$

and

$$(\lambda x.M =_S \lambda y.P) \Rightarrow ((x =_S y) \wedge (M =_S P)).$$

The λ-calculus is said to be pure, if the set of constants is empty, i.e., $C = \emptyset$, otherwise it is said to be applied.

In the pure λ-calculus, we have only functions of the form $\lambda x.M$. The substitution of free occurrences of bound variables in functions is its sole legitimate operation (or transformation). Nevertheless, the pure λ-calculus constitutes a complete formal model which provides the bare essentials for a formal reasoning about abstract algorithms. Numbers, truth values, even list structures, as well as value-transforming or structuring functions to be performed on them can be represented as (admittedly rather complicated, hardly intelligible) pure λ-terms.

In the sequel we will be primarily concerned with the concept of substitution. It therefore suffices to consider only the pure λ-calculus even though it is way too primitive to study non-trivial computational problems in a comprehensible form.

5.3.1 Reduction

We learned that the application $(\lambda x.M\ N)$ is supposed to represent the result of applying the operator $\lambda x.M$ to the operand N. This result must be computed by the substitution of free occurrences of the variable x in M by N, denoted as $M[x \Leftarrow N]$.

The transformation

$$(\lambda x.M\ N) \longrightarrow_\beta M[x \Leftarrow N]$$

[12] Note that we are in a conflict here with respect to the proper terminology. In applicative languages, applicators are considered the sole (universal) operators, and its components are considered the operands. This complies with the terminology used in chapter 4. However, the λ-calculus distinguishes between operators and operands with respect to the components of applications (which are implicitly represented as pairs of brackets). In the following, we will use the terms operator and operand as in the λ-calculus. Alternatively, we will refer, somewhat sloppily, to operators as functions and to operands as arguments as well.

is called β-reduction or β-contraction. It is said to simplify or reduce the β-redex $(\lambda x.M\ N)$ to its contractum $M[x \leftarrow N]$.

In order to formally define substitution, we first need to define what is meant by the binding status of a variable in the context of the λ-calculus.

Let M be a λ-term, and let $FV(M)$ be the set of free variables, $BV(M)$ the set of bound variables in M. Then we have

$$FV(M) = \begin{cases} \{x\} & \text{if } M =_S x \in V \\ FV(P) \cup FV(Q) & \text{if } M =_S (P\ Q) \\ FV(P) \setminus \{x\} & \text{if } M =_S \lambda x.P \end{cases}$$

$$BV(M) = \begin{cases} \emptyset & \text{if } M =_S x \in V \\ BV(P) \cup BV(Q) & \text{if } M =_S (P\ Q) \\ BV(P) \cup \{x\} & \text{if } M =_S \lambda x.P \end{cases}$$

A variable x in a λ-term M is said to be free iff $x \in FV(M)$ and bound iff $x \in BV(M)$. In a λ-term $\lambda x.M$, the term M is called the scope of the binder λx.

In compliance with our informal definition of open and closed functions in section 5.1, we say that a λ-term M is closed (or a combinator) iff $FV(M) = \emptyset$, otherwise it is open.

Examples of combinators which we will use later on include

$$\begin{array}{lcl} \Omega & =_S & \lambda x.(x\ x) \\ S & =_S & \lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z)) \\ K & =_S & \lambda x.\lambda y.x \\ I & =_S & \lambda x.x\ . \end{array}$$

The combinators S, K, I were independently introduced by Schoenfinkel [Scho24] and by Curry [Curr29, Curr34, Curr36] as a basis for what is called combinatory logic. They are computationally complete in the sense that they suffice to represent all computable functions: the combinator I reproduces its argument, the combinator K reproduces its first and consumes its second argument, and the combinator S duplicates its third argument. Thus, we have an identity operation, as well as annihilation and duplication of λ-terms. In fact, the combinator I is even redundant for it can be equivalently denoted as SKK.

We are now ready to define the substitution by a λ-term N of free occurrences of a variable x in a λ-term M by β-reduction as follows:

$$M[x \Leftarrow N] = \begin{cases} N & \text{if } M =_S x \in V \\ h & \text{if } M =_S h \in V \backslash \{x\} \\ (P[x \Leftarrow N]\ Q[x \Leftarrow N]) & \text{if } M =_S (P\ Q) \\ \lambda x.P & \text{if } M =_S \lambda x.P \\ \lambda y.P[x \Leftarrow N] & \text{if } M =_S \lambda y.P \\ & \text{and } (y \notin FV(N)) \vee (x \notin FV(P)) \\ \lambda v.(P[y \Leftarrow v])[x \Leftarrow N] & \text{if } M =_S \lambda y.P \\ & \text{and } (y \in FV(N)) \wedge (x \in FV(P)) \\ & \text{where } v \in V \\ & \text{and } (v \notin FV(P) \cup FV(N))\ . \end{cases}$$

The most important case of this definition is the last one: in order to avoid naming conflicts between free occurrences of the variable y in N and P, occurrences of y in P are renamed by substituting another variable $v \neq y$ for them which occurs neither in P nor in N.

If we were to perform a naive substitution

$$(\lambda y.P)[x \Leftarrow N] =_S \lambda y.(P[x \Leftarrow N])$$

in case of a name clash, we would obtain a parasitic binding of free occurrences of y in N by λy.

The consequences of such name clashes have been extensively discussed in subsection 5.1.2. In the context of the λ-calculus, this problem may be illustrated again by a simple example: the λ-term $\lambda y.x$ is a constant function: when applied to whatever argument, it returns the variable x as the function value. A naive substitution of x in $\lambda y.x$ by N would yield:

$$(\lambda y.x)[x \Leftarrow N] =_S \lambda y.N.$$

If $N =_S w \in V$, then we get $\lambda y.w$ which, again, is a constant function.
However, if $N =_S y \in V$, we get the identity combinator $\lambda y.y$ which reproduces its argument rather than consuming it. Thus, the meaning of the function has changed as a consequence of changing the status of the variable y from free to (parasitically) bound.

The renaming of bound variables in λ-terms by the transformation

$$\lambda y.P \longrightarrow_\alpha \lambda v.(P[y \Leftarrow v])$$

which the substitution rule requires in the case of name clashes is called α-conversion. It derives from β-reducing the application

$$\begin{array}{c}(\lambda u.\lambda v.(u\ v)\ \ \lambda y.P)\\ \downarrow_\beta\\ \lambda v.(\lambda y.P\ v)\\ \downarrow_\beta\\ \lambda v.(P[y \Leftarrow v])\ ,\end{array}$$

where $\lambda u.\lambda v.(u\ v)$ is said to be the α-conversion function.

The renaming of variables in case of name clashes proves to be the most difficult and complex part in mechanizing the β-reduction

$$(\lambda x.M\ N) \longrightarrow_\beta M[x \Leftarrow N]\ .$$

For every binder penetrated in M, the mechanism must find out whether

- the variable bound by this binder appears free in N, and whether
- there is a free occurrence of the variable to be substituted in its scope,

in which case an α-conversion must be performed. The new variable name must be generated by the system in some unique way, e.g., based on the enumeration of one name by consecutive indices derived from a counting device which is being incremented on each renaming instance.

The complexity of substitution can be substantially reduced if all bound variables are uniquely renamed à priori, again following some global enumeration scheme which assigns static indices, or if all open λ-terms are converted into closed λ-terms (or combinators) by the abstraction of free variables. In either case, the potential for name clashes disappears completely at the expense of modifying the representation of the entire λ-expression without, of course, changing its meaning.

A more elegant solution to the naming problem which preserves both the binding status and the names of variables may be based on an unbinding concept which is a symmetric complement to λ-binding.

Variable occurrences in a λ-term may be preceded by sequences of some i unbinding operators $\backslash$ (called λ-bars) to protect them against the innermost i binders for the particular variable name.

Let $\underbrace{\backslash\backslash \ldots \backslash}_{i} x = \backslash_{(i)}\ x$ ($i \in \{0, 1, \ldots\}$) denote an occurrence of the variable x with i-fold protection, and let $j \in \{0,\ 1, \ldots\}$ enumerate nested binders λx, counted from the variable occurrence outwards. Then, with respect to the j-th binder λx, an occurrence $\backslash_{(i)}\ x$ in the body expression is said to be

• protected if $i > j$, • free if $i = j$, • bound if $i < j$.

The index i in fact identifies the binding level (or binding status) of the variable occurrence. Protected variables in a λ-expression which are bound higher up are also referred to as relatively free variables.

The β-reduction of an application $(\lambda x.M\ N)$ can now be defined as follows: an occurrence of $\backslash_{(i)}\ x$ in

- the function body term M
  - decrements its protection index i by one if it is protected;
  - is substituted by the argument term N if it is free;
  - remains unchanged if it is bound;
- the argument term N
  - increments its protection index by l if it is free or protected,
  - remains unchanged if it is bound,

  when substituting it into the scope of some $l > 0$ nested binders λx within the term M.

As an example, we consider the β-reduction of the λ-term

$$(\lambda u.\underbrace{(\lambda w.(\ldots u \ldots w \ldots (\lambda w.(\ldots u \ldots w \ldots \backslash w \ldots)\ a))\ b)}_{M}\ \underbrace{w}_{N})\ .$$

Its outermost application is supposed to substitute a free occurrence of the variable w in operand position for free occurrences of u in the function body expression M, thereby penetrating the scope of up to two nested binders λw against which the operand w must be protected in order to maintain its binding status. When reducing the redexes from outermost to innermost, we obtain the sequence

$$\begin{array}{c}
(\lambda u.(\lambda w.(\ldots u \ldots w \ldots (\lambda w.(\ldots u \ldots w \ldots \backslash w \ldots)\ a))\ b)\ w) \\
\downarrow_\beta \\
(\lambda w.(\ldots \backslash w \ldots w \ldots (\lambda w.(\ldots \backslash\backslash w \ldots w \ldots \backslash w \ldots)\ a))\ b) \\
\downarrow_\beta \\
(\ldots w \ldots b \ldots (\lambda w.(\ldots \backslash w \ldots w \ldots b \ldots)\ a)) \\
\downarrow_\beta \\
(\ldots w \ldots b \ldots (\ldots w \ldots a \ldots b \ldots))\ .
\end{array}$$

In the resulting redex-free expression, all occurrences of w which have been substituted for u are free again.

The advantages of this unbinding concept are twofold:

- the protection indices realize a dynamic enumeration scheme for occurrences of identically named bound variables which uniquely identifies their binding levels, rather than renaming à priori all bound variables globally, following some static enumeration scheme;
- in the course of β-reductions, binding levels change dynamically based on simple, locally applicable decrement- and increment-rules, which facilitates the mechanization in a machine.

Free occurrences of variables in a λ-term merely serve as place-holders (or dummies) for λ-terms which, by β-reduction, are substituted for them. We are therefore free to rename bound variables by α-conversion without changing the meaning of the λ-term. What really matters is the binding level of a variable occurrence.

Since binding levels of occurrences of identically named variables can be uniquely identified by protection indices, we may

- systematically rename all bound variables into one variable name by α-conversion, and subsequently
- drop the variable name altogether,

thereby obtaining a λ-term with nameless binders, and with nameless protection indices (or binding levels) in the places of variable occurrences.

We will refer to these indices (in reference to their inventors) as Berkling/de Bruijn indices, or as number variables [Brui72, BeFe82a, BeFe82b].

Consider, as an example, the λ-term

$$\lambda x.(\lambda x.\lambda y.(\lambda z.((x\ y)\ z)\ (y\ x))\ \lambda y.\ (y\ x))$$

in which the binding structure is represented by arrows pointing from the binders to the respective variable occurrences.

When applying the α-conversion function $\lambda u.\lambda v.(u\ v)$ to, say, all λy first, we obtain:

$$\begin{gathered}
\lambda x.(\lambda x.(\lambda u.\lambda v.(u\ v)\ \lambda y.(\lambda z.((x\ y)\ z)\ (y\ x)))\ (\lambda u.\lambda v.(u\ v)\ \lambda y.(y\ x))) \\
\downarrow_\beta \\
\lambda x.(\lambda x.\lambda v.(\lambda y.(\lambda z.((x\ y)\ z)\ (y\ x))\ v)\ \lambda v.(\lambda y.(y\ x)\ v)) \\
\downarrow_\beta \\
\lambda x.(\lambda x.\lambda v.(\lambda z.((x\ v)\ z)\ (v\ x))\ \lambda v.(v\ x))
\end{gathered}$$

Applying the α-conversion to λz we get:

$$\begin{array}{c}
\lambda x.(\lambda x.\lambda v.((\lambda u.\lambda v.(u\ v)\ \lambda z.((x\ v)\ z))\ (v\ x))\ \lambda v.(v\ x)) \\
\downarrow_\beta \\
\lambda x.(\lambda x.\lambda v.(\lambda v.(\lambda z.((x\ \ \backslash v)\ z)\ v)\ (v\ x))\ \lambda v.(v\ x)) \\
\downarrow_\beta \\
\lambda x.(\lambda x.\lambda v.(\lambda v.((x\ \ \backslash v)\ v)\ (v\ x))\ \lambda v.(v\ x))
\end{array}$$

Applying the α-conversion to the innermost λx yields

$$\begin{array}{c}
\lambda x.((\lambda u.\lambda v.(u\ v)\ \lambda x.\lambda v.(\lambda v.((x\ \ \backslash v)\ v)\ (v\ x)))\ \lambda v.(v\ x)) \\
\downarrow_\beta \\
\lambda x.(\lambda v.\lambda v.(\lambda v.((\backslash\backslash\ v\ \ \backslash v)\ v)\ (v\ \ \backslash v))\ \lambda v.(v\ x))
\end{array}$$

and, finally, to the outermost λx, we get

$$\lambda v.(\lambda v.\lambda v.(\lambda v.((\backslash\backslash\ v\ \ \backslash v)\ v)\ (v\ \ \backslash v))\ \lambda v.(v\ \ \backslash v))$$

If we now drop all variables and introduce nameless binding symbols Λ, we obtain

$$\Lambda.(\Lambda.\Lambda.(\Lambda.((\#2\ \#1)\ \#0)\ (\#0\ \#1))\ \Lambda.(\#0\ \#1))$$

where all $\#i$ represent Berkling/de Bruijn indices (or number variables) which, in terms of binding levels, exhibit the same binding structure as the original λ-term.

The β-reduction of redexes in this term from outermost to innermost yields the transformation steps

$$\begin{array}{c}
\downarrow_\beta \\
\Lambda.\Lambda.(\Lambda.((\Lambda.(\#0\ \#3)\ \#1)\ \#0)\ (\#0\ \Lambda.(\#0\ \#2))) \\
\downarrow_\beta \\
\Lambda.\Lambda.((\Lambda.(\#0\ \#2)\ \#0)\ (\#0\ \Lambda.(\#0\ \#2))) \\
\downarrow_\beta \\
\Lambda.\Lambda.((\#0\ \#1)\ (\#0\ \Lambda.(\#0\ \#2)))\ .
\end{array}$$

Particularly interesting with respect to the transformation into Berkling/de Bruijn indices are closed λ-terms (or combinators).

Let $\lambda z_1.\ \ldots \lambda z_i.\ \ldots \lambda z_n.expr$ denote a closed λ-abstraction,i.e., its body *expr* contains no free variables other than $z_1, \ldots, z_n$. It is called a supercombinator iff all

λ-abstractions that occur in its body are closed as well [Hugh82, John85, Peyt87]. By α-conversion of the bound variables $z_1, \ldots, z_n$ of this supercombinator into v we get

$$\lambda v.\ \ldots \lambda v.expr'$$

where $expr'$ is obtained from $expr$ by replacing, for all $i \in \{1, \ldots, n\}$, free occurrences of the variable z_i by $\backslash_{(n-i)}\ v$. After dropping v, we have

$$\underbrace{\Lambda \ldots \Lambda}_{n}.\#expr = \Lambda_{(n)}.\#expr\ ,$$

where $\Lambda_{(n)}$ denotes a sequence of n nameless binders, and $\#expr$ derives from $expr'$ by the substitution $\backslash_{(j)}\ v \Leftarrow \#j$ for all $j \in \{0, \ldots, n-1\}$.

Thus, the binding levels represented by the de Bruijn indices in $\#expr$ are from the interval $[0\,..\,(n-1)]$, where n is the number of binders. When β-reducing the supercombinator application

$$(\ldots((\Lambda_{(n)}.\#expr\ N_1)\ N_2)\ldots N_r),\ (r \geq n)\ ,$$

systematically from innermost to outermost, the binders Λ are consumed from left to right so that the binding indices in $\#expr$ never change: they represent a static binding structure in which, for all $i \in \{1, \ldots, r\}$, occurrences of the de Bruijn indices $\#(n-i)$ will be substituted by the respective operand terms N_i.

It is also important to note that free binding indices which may occur in the operand terms N_i will not have changed once all β-reductions are done. This is due to the fact that all Λ-binders whose scopes may have been penetrated by β-reductions have then disappeared.

Consider as a simple example the stepwise reduction of the supercombinator application

$$\begin{array}{c}
(((\Lambda\Lambda\Lambda.(\#2\ \#1\ \#0)\ (\#2\ \#3))\ b)\ a) \\
\downarrow_\beta \\
((\Lambda\Lambda.((\#4\ \#5)\ \#1\ \#0)\ b)\ a) \\
\downarrow_\beta \\
(\Lambda.((\#3\ \#4)\ b\ \#0)\ a) \\
\downarrow_\beta \\
((\#2\ \#3)\ b\ a)
\end{array}$$

Thus, when β-reducing supercombinator applications to full argument sets in one conceptual step, all substitutions can in fact be done naively.

The mechanization of this substitution scheme in a graph reduction system becomes extremely simple: pointers to λ-terms in operand positions may in their order of application, simply be pushed into a stack; occurrences of de Bruijn indices within the supercombinator body are taken as offsets relative to the top of this stack to access the particular operand terms which must be substituted in their places.

We will see later on that supercombinators and supercombinator reductions play an important role in implementing functional languages.

5.3.2 Reduction Sequences and the Church-Rosser Property

With a precise definition of β-reduction at hand, we will now have a closer look at some important properties of reduction sequences.

If P is a λ-term which contains a redex $(\lambda x.M\ N)$, and P' is a λ-term which emerges from P by the β-reduction of this redex, then P is said to be β-reducible to P' in one step, denoted as

$$P \longmapsto_\beta P' \ ,$$

A λ-term P is β-reducible to a λ-term Q, denoted as

$$P \Vdash\!\!\longrightarrow_\beta Q \ ,$$

iff P can be transformed into Q by a finite sequence of β-reductions and α-conversions.

Based on these definitions, we can formalize what is meant by the semantic equivalence of λ-terms: two λ-terms P and Q are semantically equivalent (or have the same meaning), denoted as $P = Q$, iff P can be transformed into Q by a finite, possibly empty sequence of β-reductions or reverse β-reductions and α-conversions. This is to say that there ought to exist a finite sequence of λ-terms $P_0, \ \ldots, P_i, \ \ldots, \ P_n$ $(n \geq 0)$ such that $P_0 = P$ and $P_n = Q$ and

$$(\forall i \in \{0, \ldots, n-1\}) \ ((P_i \ \longmapsto_\beta P_{i+1}) \lor (P_{i+1} \ \longmapsto_\beta P_i) \lor (P_i \longrightarrow_\alpha P_{i+1})).$$

We say that a λ-term is in normal form (or β-normal form) iff it contains no β-redexes. If a λ-term P is β-reducible to Q, and Q is in normal form, then Q is said to be the normal form ($N\!F$) of P.

To exhibit some basic properties of reduction sequences, we consider as a first example the reduction of the λ-term $(\lambda x.(\lambda y.(x\ (x\ y))\ u)\ v)$. It may proceed along two different paths, as the redexes can be reduced in any order:

$$
\begin{tikzcd}
 & (\lambda x.(\lambda y.(x\ (x\ y))\ u)\ v) \arrow[dl, mapsto, "\beta"'] \arrow[dr, mapsto, "\beta"] & \\
(\lambda y.(v\ (v\ y))\ u) \arrow[dr, mapsto, "\beta"'] & & (\lambda x.(x\ (x\ u))\ v) \arrow[dl, mapsto, "\beta"] \\
 & (v\ (v\ u)) &
\end{tikzcd}
$$

We note that both reduction sequences terminate with the λ-term $(v\ (v\ u))$, which is the normal form of all λ-terms preceding it. Moreover, all λ-terms have the same meaning (or are semantically equivalent), as they can obviously be transformed into each other by at most two β-reductions (or reverse β-reductions).

As a second example, let us consider the reduction of the application $(\Omega\ \Omega) =_S (\lambda x.(x\ x)\ \lambda x.(x\ x))$. It causes a non-terminating sequence of β-reductions as this application incessantly reproduces itself:

$$
\begin{array}{c}
(\lambda x.(x\ x)\ \lambda x.(x\ x)) \\
\big\downarrow{\scriptstyle \beta} \\
(\lambda x.(x\ x)\ \lambda x.(x\ x)) \\
\big\downarrow{\scriptstyle \beta} \\
(\lambda x.(x\ x)\ \lambda x.(x\ x)) \\
\vdots
\end{array}
$$

Non-terminating sequences never produce a normal form. All λ-terms of such a sequence have the same meaning. However, the meaning, as we will learn later, is undefined.

The discussion of these two examples raises two interesting questions: given some λ-term,

- does it have a normal form at all, i.e., is there at least one reduction sequence which terminates after some finite number of steps?
- do all terminating sequences produce a unique normal form?

These questions, which have already been brought up in section 5.1.3, essentially relate to the existence of a function value for a function application.

The existence of a normal form for a given λ-term is generally not decidable, as we have already stated earlier. It would mean that there is some way of finding out whether or not after some finite number of β-reductions a normal form can be reached. Unfortunately, an upper limit for such a number cannot be specified [Tur36, Chur36, ChRo36].

However, the uniqueness of normal forms is guaranteed by the most important theorem of the λ-calculus, the Church-Rosser theorem [ChRo36]: it states that for any three λ-terms M, P, Q we have

$$((M \longmapsto_\beta P) \wedge (M \longmapsto_\beta Q)) \;\Rightarrow\; (\exists R)\;((P \longmapsto_\beta R) \wedge (Q \longmapsto_\beta R))$$

where R is also a λ-term (see also section 5.2).

The essence of this theorem may be illustrated by the following simple diagram (which reflects the same structure as the one for the β-reduction of the λ-term $(\lambda x.(\lambda y.(x\ (x\ y))\ u)\ v)$):

$$\begin{tikzcd} & M \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\ P \arrow[dr, "\beta"'] & & Q \arrow[dl, "\beta"] \\ & R & \end{tikzcd}$$

If any two λ-terms P and Q can be deduced from a λ-term M by finite sequences of β-reductions, then there exists another λ-term R, to which P and Q can be reduced. P and Q must not necessarily be syntactically different, and R must not necessarily be a normal form since none may exist.

We omit the proof of this theorem for its length. A recent version which has been accepted as being correct may be found in [HiSe86].

From this theorem we can immediately conclude that P and Q are α-convertible, denoted as $P =_\alpha Q$, if they are both normal forms of M.

This follows from the fact that normal forms do not contain redexes. Hence, P and Q must be α-convertible into R, which is also in normal form, and therefore be α-convertible into each other. Put differently: disregarding α-conversions, any two terminating sequences of β-reductions applied to a λ-term M always produce a unique normal form and thus realize strictly functional computations.

In another form, the Church-Rosser theorem states that for any two λ-terms P, Q we have

$$(P = Q) \Rightarrow\; (\exists\, N)\;((P \longmapsto_\beta N)\;\wedge\;(Q \longmapsto_\beta N))\;,$$

i.e., the semantic equivalence of two λ-terms P and Q implies the existence of a third λ-term N into which both can be transformed by finite sequences of β-reductions and α-conversions.

This theorem can be rather straightforwardly proved by induction over the sequence of λ-terms $P_0, \ldots, P_{n-1}, P_n$ via which $P = P_0$ can be transformed into $Q = P_n$, and vice versa [Bar81, HiSe86].

$$\begin{tikzcd}
P_0 \arrow[r, leftrightarrow] \arrow[drr, "\beta"'] & \cdots \arrow[r, leftrightarrow] & P_{n-1} \arrow[r, leftrightarrow] \arrow[d, "\beta"] & P_n \arrow[d, "\beta"] \\
 & & N' \arrow[r, mapsto, "\beta"'] & N''
\end{tikzcd}$$

For $n = 0$, the proof is trivial. For the induction step from $n-1$ to n, we assume that either $P_{n-1} \longmapsto_\beta P_n$ or $P_n \longmapsto_\beta P_{n-1}$, and that there exist λ-terms N' and N'' such that $P_0 \Vdash\!\!\longrightarrow_\beta N'$, $P_{n-1} \Vdash\!\!\longrightarrow_\beta N'$ and $N' \longmapsto_\beta N''$, $P_n \Vdash\!\!\longrightarrow_\beta N''$.

If $P_n \longmapsto_\beta P_{n-1}$, then we also have $P_n \Vdash\!\!\longrightarrow_\beta N'$ and, hence, $N = N'$.

If, however, $P_{n-1} \longmapsto_\beta P_n$, then we apply the Church-Rosser theorem in its original form, with $P_{n-1},\ N', P_n,\ N''$ substituting for $M,\ P,\ Q,\ R$, respectively, and obtain $N = N''$.

Following this theorem, it is easy to see that P can be β-reduced to Q if P and Q are semantically equivalent, and Q is a normal form.

Furthermore, if P and Q are semantically equivalent, then both have either the same normal form, or no normal form at all.

This proposition may be verified by the following consideration: let $\overline{P}$ and $\overline{Q}$ be the normal forms of P and Q, respectively. From $P = Q$ follows $\overline{P} = \overline{Q}$ and $P \Vdash\!\!\longrightarrow_\beta \overline{Q}$. Since $\overline{P}$ is in normal form, we also have $\overline{P} =_\alpha \overline{Q}$.

Suppose now there exists a normal form $\overline{Q}$ for Q but none for P. From $P = Q$ and $Q \Vdash\!\!\longrightarrow_\beta \overline{Q}$ follows, however, that $P = \overline{Q}$ and, hence, $P \Vdash\!\!\longrightarrow_\beta \overline{Q}$, which contradicts our supposition.

5.3.3 Reduction Disciplines

The objective of transforming λ-terms by sequences of β-reductions is to deduce their normal forms (if they at all exist) in order to represent their meanings in the simplest possible form. In general, λ-terms may contain several redexes which may be reduced in any order, i.e., we have a transformation relation $\tau \subset E_\lambda \times E_\lambda$

which is total in its left component. For a λ-term that has a meaning, however, the Church-Rosser property guarantees that all terminating sequences of β-reductions yield a unique normal form.

With respect to a redex $(\lambda x.M\ N)$ we basically have the choice between two alternative reduction disciplines.

Applicative order reduction requires that the operand λ-term N be reduced to its normal form before the operator $\lambda x.M$ is applied to it. The idea is to transform N exactly once before its substitution for possibly many occurrences of the variable x in the λ-term M. It corresponds to a call-by-value parameter passing discipline in conventional programming languages.

When applying this discipline recursively to all subterms of a λ-term, say R, the transformation relation τ becomes a function $\tau_A \subset E_\lambda \times E_\lambda$ which is defined as follows:

$$\tau_A(R) = \begin{cases} x & \text{if } R =_S x \in V \\ \lambda x.\tau_A(P) & \text{if } R =_S \lambda x.P \\ \tau'_A(\tau_A(P)\ \tau_A(Q)) & \text{if } R =_S (P\ Q) \text{ and } P \neq_S \lambda x.P' \\ \tau'_A(P\ \tau_A(Q)) & \text{if } R =_S (P\ Q) \text{ and } P =_S \lambda x.P' \end{cases}$$

where

$$\tau'_A(R) = \begin{cases} \tau_A(P[x \Leftarrow Q]) & \text{if } R =_S (\lambda x.P\ Q) \\ (P\ Q) & \text{if } R =_S (P\ Q) \end{cases}$$

The function τ_A is recursively driven from outermost to innermost into a λ-term. It substitutes operands in normal form, if they can be produced, and generates non-terminating β-reductions otherwise.

Normal order reduction of a redex $(\lambda x.M\ N)$ requires that the operator be applied to the operand as it is, with the consequence that the λ-term N may have to be reduced several times inside the λ-term M. It corresponds to a call-by-name parameter passing discipline in conventional programming languages.

Normal order reduction applied to a λ-term R is recursively defined by the function $\tau_N \subset E_\lambda \times E_\lambda$ as follows:

$$\tau_N(R) = \begin{cases} x & \text{if } R =_S x \in V \\ \lambda x.\tau_N(P) & \text{if } R =_S \lambda x.P \\ \tau'_N(\tau_N(P)\ Q) & \text{if } R =_S (P\ Q) \text{ and } P \neq_S \lambda x.P' \\ \tau'_N(P\ Q) & \text{if } R =_S (P\ Q) \text{ and } P =_S \lambda x.P' \end{cases}$$

where

$$\tau'_N(R) = \begin{cases} \tau_N(P[x \Leftarrow Q]) & \text{if } R =_S (\lambda x.P\ Q) \\ (P\ \tau_N(Q)) & \text{if } R =_S (P\ Q) \end{cases}$$

It recursively enforces the reduction of the leftmost redex within a λ-term.

The difference between these two reduction disciplines which really matters is that

- applicative order reduction attempts to reduce operands to their normal forms irrespective of whether or not they are being inserted into operators;
- normal order reduction attempts to generate normal forms of operands only if they contribute to the normal forms of the λ-terms they are part of.

Hence, normal order reduction applied to λ-terms that have a meaning is bound to yield normal forms, while applicative order reduction may fail to do so by getting trapped in non-terminating substitutions.

The existence of terminating and non-terminating sequences of β-reductions, as we already learned in the preceding subsection, is always caused by redexes of the form

$$K\ A\ B =_S \lambda x.\lambda y.x\ A\ B \qquad \text{or} \qquad \overline{K}\ A\ B =_S \lambda x.\lambda y.y\ A\ B\ ,$$

if the λ-term B in the first case and the λ-term A in the second case contains at least one redex which has no normal form.

Applicative order reduction would get trapped in endless transformations of B and A, respectively, while normal order reduction would perform the transformations

$$K\ A\ B \longmapsto_\beta A \qquad \text{and} \qquad \overline{K}\ A\ B \longmapsto_\beta B\ .$$

Neither of these disciplines is optimal in the sense that they produce normal forms with the least number of β-reductions. Applicative order reduction may inflict redundant transformations if the normal form of an operand λ-term is consumed by the operator without reproduction; normal order reduction may inflict redundant transformations if the operand λ-term is reproduced more than once within the operator.

Besides these two basic reduction disciplines, there are variants of normal order reduction which are called lazy and fully lazy evaluation. They essentially avoid the repeated transformation of several copies of the same operand λ-term inside an operator by sharing its reduction to normal form. This, however, is an implementation principle which is outside the scope of the λ-calculus and will, therefore, not be treated in this context.

5.3.4 Recursion

Recursion constitutes a fundamental concept of the λ-calculus. It is essentially based on the reproduction of applications of λ-terms to themselves by means of β-reductions (for which the redex $(\Omega\ \Omega) =_S (\lambda x.(x\ x)\ \lambda x.(x\ x))$ is a simple example).

The formal treatment of recursion in the λ-calculus derives from the so-called fixed-point theorem.

It states that for all λ-terms F there exists a λ-term Q such that $(F\ Q) = Q$, where Q is the fixed point of F.

A closed λ-term M is said to be a fixed-point combinator iff for all λ-terms F we have

$$(M\ F) = (F(M\ F)),$$

i.e., $(M\ F)$ is the fixed point of F.

This equation can be solved by the so-called Y-combinator, one valid form of which is given as

$$Y =_S (P\ P) \text{ where } P =_S \lambda x.\lambda y.(y\ ((x\ x)\ y))\ ,$$

since

$$\begin{array}{lll} (Y\ F) & =_S & ((\lambda x.\lambda y.(y\ ((x\ x)\ y))\ P)\ F) \\ & = & (\lambda y.(y\ ((P\ P)\ y))\ F) = (F\ ((P\ P)\ F)) = (F\ (Y\ F))\ . \end{array}$$

The Y-combinator applies, by substitution of the bound variable x, the combinator P to itself and reproduces, by substitution of the variable y, the λ-term F which is assumed to represent a recursive function.

Consider, as an example, the λ-term

$$F =_S \lambda x.\lambda y.(((F\ x)\ y)\ ((F\ y)\ x))$$

which is recursively defined by itself. We may rewrite this equation as

$$F = (\lambda f.R\ \ F)$$

with $R =_S \lambda x.\lambda y.(((f\ x)\ y)\ ((f\ y)\ x))$.

Following the definition of the fixed-point combinator Y, a solution for this equation must be of the form

$$F = (Y\ \lambda f.R)$$

in order to get

$$(Y\ \lambda f.R) = (\lambda f.R\ (Y\ \lambda f.R))\ .$$

In general, we have the following proposition: let R be a λ-term containing free occurrence of the variable f, then there exists a λ-term F such that

- $F = (\lambda f.R\ \ F)$
- $F \longmapsto_\beta R[f \Leftarrow F]$.

The proof is trivial.

This proposition specifies all recursive functions F as fixed points of abstractions of the form $\lambda f.R$, and the β-reduction $F \longmapsto_\beta R[f \Leftarrow F]$ as a rule of computation for recursive functions.

The universal character of the Y-combinator essentially derives from the fact that it reproduces the abstraction $\lambda f.R$ as a by-product of reproducing itself on each recursion step. The additional β-reductions that reproduce the Y-combinator can, however, be avoided if recursion is realized by more direct means.

Rather than transforming a recursive function definition F into $F = (Y\ \lambda f.R)$, we could simply define F to be an application of the form

$$F = (F^* F^*) \text{ with } F^* = \lambda f.R^*,$$

where R^* derives from R in that all free occurrences of f are being replaced by applications $(f\ f)$.

It can easily be verified that the redex $(F^*\ F^*)$ effects, by means of a single β-reduction, the direct reproduction of F in itself.

In the case of the above example we have

$$F = \lambda x.\lambda y.(((F\ x)\ y)\ ((F\ y)\ x)) = (F^* F^*)$$

with $F^* = \lambda f.\lambda x.\lambda y.((((f\ f)\ x)\ y)\ (((f\ f)\ y)\ x))$.

When extending the λ-calculus by a special recursion operator α, recursive λ-terms may be specified even more concisely as

$$F = \alpha f.R\ .$$

The recursive binder αf is to effect the direct substitution of free occurrences of f in R in a way that is fully equivalent to β-reducing the application

$$(\lambda f.R\ \alpha f.R) \longmapsto_\beta R[f \Leftarrow \alpha f.R]\ .$$

Other than avoiding superfluous β-reductions, the recursion operator α also solves another rather annoying problem: the Y-combinator $Y =_S (P\ P)$ cannot be used under an applicative order reduction regime. Following its definition in the preceding subsection, it can easily be verified that under this regime the self-application $(P\ P)$ engages in endless recursions for it reproduces itself incessantly in the operand position of an application. The α-operator prevents this by the direct substitution of $\alpha f.R$ in itself in unreduced form, which is equivalent to reducing the Y-combinator in normal order.

Sets of mutually recursive functions which have the general form

$$\begin{aligned}
F_1 \quad &= \ldots F_1 \ldots F_i \ldots F_k \ldots \\
&\vdots \\
F_i \quad &= \ldots F_1 \ldots F_i \ldots F_k \ldots \\
&\vdots \\
F_k \quad &= \ldots F_1 \ldots F_i \ldots F_k \ldots
\end{aligned}$$

may be transformed into λ-terms in a similar way. When abstracting, say from F_i, the function names $F_1, \ldots,\ F_i, \ldots,\ F_k$, we obtain

$$F_i = (\lambda\vec{f}.R_i\ F_1 \ldots F_i \ldots F_k)$$

where $\lambda\vec{f}.\ R_i = \lambda f_1.\ \ldots\lambda f_i.\ \ldots\lambda f_k.\ \dashv \ldots f_1 \ldots f_i \ldots f_k \ldots \vdash$ (with $\dashv\ \vdash$ denoting the delimiters of the body expression R_i). When abstracting F_i again, we get

$$F_i = (\lambda f_i.(\lambda\vec{f}.R_i\ F_1 \ldots f_i \ldots F_k)\ F_i)$$

and as a solution for this equation,

$$F_i = (Y\ \lambda f_i.(\lambda\vec{f}.R_i\ F_1 \ldots f_i \ldots F_k))\ .$$

Using the recursion operator α instead of the Y-combinator, we have

$$F_i = \alpha f_i.(\lambda\vec{f}.R_i\ F_1 \ldots f_i \ldots F_k)\ .$$

In either case, the λ-term on the right-hand side reduces to

$$\begin{aligned}
F_i \quad &= (\lambda\vec{f}.R_i\ F_1 \ldots (Y\ \lambda f_i.(\lambda\vec{f}.R_i\ F_1 \ldots f_i \ldots F_k)) \ldots F_k) \\
&= (\lambda\vec{f}.R_i\ F_1 \ldots F_i \ldots F_k)
\end{aligned}$$

(after three β-reductions)

or to

$$\begin{aligned}
F_i \quad &= (\lambda\vec{f}.R_i\ F_1 \ldots \alpha f_i.(\lambda\vec{f}.R_i\ F_1 \ldots f_i \ldots F_k) \ldots F_k) \\
&= (\lambda\vec{f}.R_i\ F_1 \ldots F_i \ldots F_k)
\end{aligned}$$

(after one recursion step),

respectively.

Thus, our set of mutually recursive function definitions can, in terms of λ-expressions, equivalently be represented as

$$\begin{aligned} F_1 &= (\lambda \vec{f}.R_1\ F_1 \ldots F_i \ldots F_k) \\ &\ \ \vdots \\ F_i &= (\lambda \vec{f}.R_i\ F_1 \ldots F_i \ldots F_k) \\ &\ \ \vdots \\ F_k &= (\lambda \vec{f}.R_k\ F_1 \ldots F_i \ldots F_k)\ . \end{aligned}$$

5.3.5 The Meaning of λ-Terms

Based on the notion of semantic equality of λ-terms which we introduced in subsection 5.3.2, we can formalize what is meant by the semantics of λ-terms.

A first definition was given by Church in 1941 [Chur41]: let E_λ denote the set of λ-terms, $C_\lambda \subset E_\lambda$ the set of normal forms, and $m \subset E_\lambda \times C_\lambda$ a partial function, then the semantics or meaning of a λ-term $M \in E_\lambda$ is defined as

$$m\,[\![M]\!] = \begin{cases} N \in C_\lambda & \text{if } N \text{ is the normal form of } M \\ \textit{undefined} & \text{otherwise .} \end{cases}$$

This definition causes problems with λ-terms containing non-terminating components.

Consider, as an example, the λ-term

$$K\ I\ (\Omega\ \Omega) \quad \text{with} \quad \begin{aligned} K &=_S \lambda x.\lambda y.\lambda x \\ I &=_S \lambda x.x \\ \Omega &=_S \lambda x.(x\ x) \end{aligned}$$

The existence of a normal form can only be determined by reducing the term, which can be done in two different ways:

- if we apply the outermost combinator K to its arguments as they are, we obtain a normal form after two β-reductions:

$$K\ I\ (\Omega\ \Omega) \longmapsto_\beta I$$

- however, if we try to reduce the arguments of the combinator K first, we end up with a non-terminating reduction sequence:

$$K\ I\ (\Omega\ \Omega) \longmapsto_\beta K\ I\ (\Omega\ \Omega) \longmapsto_\beta \ \ldots$$

Thus, we have λ-terms which can be reduced to normal forms. However, the normal forms can not be reached with every possible reduction sequence.

To overcome these difficulties, Church has, in a very rigid interpretation of the above definition, demanded that a λ-term has a meaning if and only if all of its subterms have a meaning (or can be transformed to normal forms). This demand is, in very restrictive form, satisfied by the terms of the so-called λI-calculus. However, it rules out all abstractions $\lambda x.M$ with $x \notin FV(M)$ and thus the K-combinator. Without it, selection among alternatives, which is essential for the specification of terminating recursions, cannot be accomplished by direct means.[13] From a pragmatic point of view, the λI-calculus is therefore not very useful.

In the extended so-called λK-calculus, a meaning is defined for all λ-terms which are solvable or have a head normal form in compliance with the following definitions [Bar81].

A closed λ-term (or combinator) M is said to be solvable iff there exist λ-terms $N_1, \ldots, N_n$ such that

$$(M\ N_1 \ldots N_n) = I\ ,$$

i.e., M applied to $N_1, \ldots, N_n$ yields the identity combinator.

An open λ-term M with the set of free variables $FV(M) = \{z_1, \ldots, z_q\}$ is said to be solvable iff

$$M' = \lambda z_1.\ \ldots \lambda z_q.M$$

is solvable.

Thus, solvability of a λ-term M requires the existence of at least one set of arguments, the one for which the function M produces the combinator I as function value.

Examples of solvable λ-terms are

- S since $S\ I\ I\ I = I$
- $\lambda x.((x\ I)\Omega)$ since $(\lambda x.((x\ \ I)\Omega)\ K) = I$
- $\lambda x.(x\ (M\ \Omega))$ since $(\lambda x.(x\ (M\ \Omega))\ (K\ I)) = I$

We consider now a λ-term of the general form

$$P =_S \lambda x_1.\ \ldots \lambda x_n.(z\ M_1 \ldots M_m)\ ;\ (m, n \geq 0,\ z \in \{x_1, \ldots, x_n\})\ .$$

[13] An abstraction equivalent to the K-combinator may be constructed from applications to I-combinators so that the λ-term in its second argument position is systematically 'eaten up', whereas the first argument term is reproduced. A simple example is the abstraction $K' =_s \lambda x.\lambda y.(((y\ I)I)x)$. The reader may convince himself that the application $((K'\ M)N)$ with $N =_s \lambda u.\lambda v.(u\ v)$ reproduces M and consumes N (see also [Bar81]).

It is easy to see that this term can only be reduced to a λ-term

$$P^* =_S \lambda x_1. \; \ldots \lambda x_n.(z \; M_1^* \ldots M_m^*)$$

with

$$(\forall i \in \{1, \ldots, m\}) \; (M_i \longmapsto_\beta M_i^*) \; .$$

We note that the head part $\lambda x_1. \; \ldots \lambda x_n.(z \ldots$ remains unchanged.

The λ-term P is therefore said to be in head normal form.

A λ-term M is said to have a head normal form iff there exists a λ-term P in head normal form and $M = P$. If a λ-term P' is not in head normal form, then it has the form

$$P' =_S \lambda x_1 \; . \; \ldots \lambda x_n. \; ((\lambda y.Q \; M_0) \; M_1 \ldots M_m) \; ; \; (m, n \geq 0) \; ,$$

where $(\lambda y.Q \; M_0)$ is called the head-redex.

This definition leads us to conclude that a λ-term has a head normal form iff there is a reduction sequence which, when applied to the head-redex, terminates.

A normal form is, of course, also a head normal form, but not necessarily vice versa. Thus, the existence of a head normal form is a decidedly weaker property than the existence of a normal form.

From the theorem which says that a λ-term is solvable iff it has head normal form we may deduce the following propositions regarding the semantics (or meaning) of terms of the λK-calculus:

- the meaning of a λ-term is undefined iff it is not solvable;
- a λ-term is not solvable iff it has no head normal form;
- a λ-term without a head normal form is also without a normal form, but not vice versa.

Using these propositions, we define the meaning m of a λ-term M in a less rigorous way than Church as

$$m \, [\![M]\!] = \begin{cases} N & \text{if it is the normal form of } M; \\ \textit{undefined} & \text{if } M \text{ has no head normal form;} \\ P & \text{else} \end{cases}$$

where P is a head normal form of M.

The pragmatism of this definition deserves some further interpretation: a λ-term M which can be reduced to a head normal form $P =_S \lambda x_1. \; \ldots \lambda x_n.(z \; M_1 \ldots M_m)$, in fact, represents a function of n variables. The solvability of this function is defined by the existence of a set of n arguments $N_1, \ldots, N_n$ for which the function produces a particular value (the identity combinator I). If the function is indeed solvable, it (also) has a non-empty domain. To verify solvability it suffices to show that the function has a head normal form. If it has one, it also is assumed to have a meaning.

Thus, the definition declares all functions meaningful which have a non-empty domain comprising at least the (sets of) argument λ-terms which yield the identity combinator as a function value.

5.3.6 An Applied λ-Calculus

We learned throughout the preceding subsections that the pure λ-calculus provides a formal model for executing abstract algorithms (or rules of computation). It allows us to exhibit and reason about basic properties of functions and function applications at a very high level of abstraction.

However, the pure λ-calculus is not suited as a practical programming language. Though it is possible to represent numbers, truth values and lists as combinator terms and to specify abstractions which effect the equivalent of arithmetic, relational, logical and structuring operations on these representations, the resulting λ-terms are difficult, if not impossible to interpret. Moreover, they tend to become extremely complex for applications of only moderate size, inflicting a multiple of β-reductions for all the primitive operations that need to be performed.

Even a primitive version of an applied λ-calculus can be expected to remedy this problem to some considerable extent. In its set of constants, it should include at least decimal numbers, character strings, the Boolean constants TRUE and FALSE, and the symbols $<,>$ for the construction of lists on the one hand, and primitive function symbols for arithmetic, logic and list operations on the other hand.

Following the recursive construction of λ-terms in the form of nested operator-operand pairs, we need to represent

- binary operations of the form $(arg\_2 \text{ OP } arg\_1)$ as $((\text{OP } arg\_1)\ arg\_2)$,
- unary operations of the form $\text{OP}(arg)$ as $(\text{OP } arg)$.

Thus, we have the equivalent representations

$$\begin{aligned}
(3 \text{ EQ } 2) &\equiv ((\text{EQ } 2)3) \\
((4-1) \ * \ (2+3)) &\equiv ((*\ ((+\ 3)\ 2))\ ((-\ 1)\ 4)) \\
(5 \text{ GT } 3) &\equiv ((\text{GT } 3)\ 5) \\
\text{FIRST}(<a,b,c>) &\equiv (\text{FIRST } <a,b,c>)\ .
\end{aligned}$$

The transformation of λ-terms that specify applications of primitive functions such as $+ \mid - \mid \ldots \mid \text{GT} \mid \text{EQ} \mid \ldots$ etc. is called δ-reduction.

Let $M, N \in L$ be terms of an applied λ-calculus, then M is said to be δ-reducible to N in one step, denoted as $M \longmapsto_\delta N$, iff

- $M =_S (a\ M_1 \ldots M_n)$, where $a \in PF \subset C$, $M_1, \ldots, M_n \in L$, and PF is the set of primitive functions;
- the λ-terms $M_1, \ldots, M_n$ are irreducible with respect to β- and δ-reductions;
- and $FV(M) = FV(N) = \emptyset$.

This definition in fact demands that δ-reductions be executed after completion of all β-reductions that affect free occurrences of bound variables in M, and strictly in applicative order. The pragmatism of this derives from the fact that all primitive functions can only be applied to arguments that are within their respective domains, to which neither variables nor applications belong.

IF-THEN-ELSE clauses have, in the applied λ-calculus, the general form $((P\ M)\ N)$. This is based on the supposition that the Boolean constants TRUE and FALSE, to either of which the predicate term P is expected to reduce, realize the combinators K and $\overline{K}$, respectively; i.e., we have

$$\begin{array}{lll} \text{TRUE} & \equiv K & =_S \lambda x.\lambda y.x \\ \text{FALSE} & \equiv \overline{K} & =_S \lambda x.\lambda y.y\ . \end{array}$$

A λ-term of the applied λ-calculus is in normal form iff it contains neither β-redexes nor δ-redexes.

We consider now, as an example for the representation and transformation of an algorithm in the applied λ-calculus, the recursive specification of the factorial function. In the notation used throughout section 5.1 of this chapter, it has the form

$$\begin{array}{rll} fac(n) = & \text{IF} & (n\ \text{GT}\ 1) \\ & \text{THEN} & (n\ *\ fac((n-1))) \\ & \text{ELSE} & 1\ . \end{array}$$

This recursive equation translates into the λ-term

$$fac =_S \lambda n.((\text{GT}\ 1\ n)\ (*\ (fac\ (-\ 1\ n))\ n)\ 1)\ .$$

By abstraction of the recursively defined variable fac we obtain

$$fac =_S (\lambda f.\lambda n.((\text{GT}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1)\ fac)\ ;$$

i.e., fac is a fixed point of the λ-term $\lambda f.\lambda n.(\ldots)$:

$$fac =_S (Y\ \lambda f.\lambda n.((\text{GT}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1))\ .$$

The application of this λ-term to the argument value 3 reduces as

$$
\begin{gathered}
(fac\ 3)\\
\Downarrow\\
((Y\ \lambda f.\lambda n.((\textsc{gt}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1))\ 3)\\
\Downarrow\\
(((\lambda x.\lambda y.(y\ ((x\ x)\ y))\ P)\ \lambda f.\lambda n.((\textsc{gt}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1))\ 3)\\
\Downarrow\\
((\lambda y.(y\ ((P\ P)\ y))\ \lambda f.\lambda n.((\textsc{gt}\ 1\ n)\ (*(f\ (-\ 1\ n))\ n)\ 1))\ 3)\\
\Downarrow\\
((\lambda f.\lambda n.(\ldots)\ (Y\ \lambda f.\lambda n.(\ldots)))\ 3)\\
\Downarrow\\
(\lambda n.((\textsc{gt}\ 1\ n)\ (*\ ((Y\ \lambda f.\lambda n.(\ldots))\ (-\ 1\ n))\ n)\ 1)\ 3)\\
\Downarrow\\
((\textsc{gt}\ 1\ 3)\ (*((Y\ \lambda f.\lambda n.(\ldots))\ (-\ 1\ 3))\ 3)\ 1)\\
\Downarrow\\
(K\ (*\ ((Y\ \lambda f.\lambda n.(\ldots))\ 2)\ 3)\ 1)\\
\Downarrow\\
(*\ ((Y\ \lambda f.\lambda n.(\ldots))\ 2)\ 3)\\
\Downarrow\\
(*\ (*\ 1\ 2)\ 3)\\
\Downarrow\\
6\ .
\end{gathered}
$$

Note that in this sequence the Y-operator is reduced in normal order while all other applications are reduced in applicative order.

When using the operator α instead of the Y-combinator, recursion can be accomplished by more direct means. The factorial function then takes the form

$$fac =_S \alpha f.\lambda n.((\textsc{gt}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1)\ ,$$

and the application to the argument value 3 proceeds as follows:

$$
\begin{gathered}
(fac\ 3)\\
\Downarrow\\
(\alpha f.\lambda n.((\textsc{gt}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1)\ 3)\\
\Downarrow\\
(\lambda n.((\textsc{gt}\ 1\ n)\ (*\ (\alpha f.\lambda n.(\ldots)\ (-\ 1\ n))\ n)\ 1)\ 3)\\
\Downarrow
\end{gathered}
$$

$$\begin{array}{c}((\textsc{gt}\ 1\ 3)\ (*\ (\alpha f.\lambda n.(\ldots)\ (-\ 1\ 3))\ 3)\ 1)\\ \Downarrow\\ \vdots\\ \Downarrow\\ 6\ .\end{array}$$

When comparing both reduction sequences, we note that it takes four β-reductions to perform recursion by means of the Y-combinator, while just one transformation equivalent to a β-reduction is necessary to achieve the same ends by means of the α-operator.

5.3.7 Brief Outline of a Typed λ-Calculus

Types introduce into programming languages the notion of functions as mappings from domain sets to range sets. This contrasts with the theory of the untyped λ-calculus which primarily deals with the operational aspects of transforming function applications. However, both views can be united within the framework of a typed λ-calculus which extends λ-terms by type expressions, if necessary [HiSe86].

Typing may be seen as a means to introduce more formal rigor into programming and to facilitate reasoning about program correctness. However, it is also of considerable practical relevance. Since almost all contemporary computing machines have no built-in type checking facilities, high-level programs must include at least to some extent type declarations in order to make sure that they are well-typed (i.e., do not produce type-incompatibilities at run-time) and to aid the compiler in transforming these programs into correct machine code.

Type expressions are recursively constructed from atomic types, function types and union types. The atomic types denote sets, say, of Boolean values, numbers, characters etc., and the function types denote mappings from domain sets to range (codomain) sets. Atomic types are denoted as $\alpha, \beta, \ldots$, function types as $(\alpha \to \beta)$, and union types as $(\alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_n)$, where $\alpha_1, \ldots, \alpha_n$ are types.

Using this basic notion of so-called monomorphic types, we can define a typed λ-calculus as follows: let x and c denote a variable and a constant other than a variable, respectively, then

- $x\!:\!\alpha$ and $c\!:\!\alpha$ are typed λ-terms of type α;
- $(M:(\alpha \to \beta)\ N:\alpha):\beta$ is a typed application of type β if M and N are typed λ-terms of the types $(\alpha \to \beta)$ and α, respectively;
- $(\lambda x:\alpha.\, M:\beta):(\alpha \to \beta)$ is a typed abstraction of type $(\alpha \to \beta)$ if x is a variable of type α and M is a λ-term of type β.

Some of the type annotations in abstractions and applications may be dropped since they can in an obvious way be inferred from other types. Likewise, if no ambiguities can occur, we may drop parentheses from nested function types and write

$$\alpha_1 \to \alpha_2 \to \ldots \alpha_{n-1} \to \alpha_n \;\text{ instead of }\; (\alpha_1 \to (\alpha_2 \to \ldots (\alpha_{n-1} \to \alpha_n) \ldots)),$$

implying association to the right.

A typed β-reduction is defined as

$$(\lambda x.\, M : (\alpha \to \beta)\; N : \alpha) \longmapsto_{\beta_t} M[x : \alpha \Leftarrow N : \alpha] : \beta \;,$$

i.e., it is carried out in the same way as a type-free β-reduction provided that the type of the operand matches the type of the λ-bound variable (or the domain type) of the operator. Otherwise, the application remains as it is.

It is important to note that in a typed β-redex the operator must be of a 'higher' type (containing at least one more type constructor $\to$) than the operand, i.e., both λ-terms cannot be the same. This rules out self-applications and thus the realization of the Y-combinator (unless we allow for recursive types). Without it, there can be no recursive λ-terms. A typed λ-calculus is therefore said to be strongly normalizing: every typed λ-term has a normal form into which it can be transformed by a finite sequence of typed β-reductions. This being the case, it can also be decided whether or not two typed λ-terms are semantically equivalent (which requires that they can be converted into each other by finitely many β-reductions (or reverse β-reductions) and α-conversions).

Strong normalization is not at all desirable from a pragmatic point of view. The absence of a recursion operator severely limits the expressive power of a typed λ-calculus. Since all functions are in fact totally defined, we can essentially compute polynomials only.

The problem with recursion is not primarily caused by typing itself, but by its implication on self-applications. We can infer a type for every fixed-point combinator μ which satisfies $(\mu\; F) = (F\; (\mu\; F))$ (see section 5.3.4), including the Y-combinator. Assuming the types α for $(\mu\; F)$ and $\alpha \to \beta$ for F, we have

$$(\mu\; F) : \alpha = (F : (\alpha \to \beta)\; (\mu\; F) : \alpha) : \beta \;.$$

By unification of the types of the λ-terms on both sides of the equation, we get $\alpha = \beta$ and subsequently

$$(\mu\; : ((\alpha \to \alpha) \to \alpha)\; F : (\alpha \to \alpha)) : \alpha = (F : (\alpha \to \alpha)\; (\mu\; F) : \alpha) : \alpha \;,$$

i.e., μ has a type $((\alpha \to \alpha) \to \alpha)$.

So, in order to support general recursions in a typed λ-calculus, it is obviously necessary to realize the fixed-point combinator μ as a primitive operator rather than by means of an explicit self-application. In fact, the recursion operator α (not to be confused with the type α) which we introduced in section 5.3.4 is exactly what we are looking for. It performs recursion by the direct substitution

$$\alpha f.R \longmapsto_{\beta} R[f \Leftarrow \alpha f.R]$$

of a term in itself.

The monomorphic types we have used so far are not too well suited for λ-terms which apply uniformly to all arguments irrespective of theit types. Typical examples are the *SKI* combinators and the recursion operator α which simply duplicated or consume their arguments. To deal with such λ-terms in the framework of a typed λ-calculus, we need to introduce the concept of polymorphic types (or type schemata). They are usually denoted as $*\alpha, *\beta, \ldots$ etc., representing propositions of the form for all types α, for all types $\beta, \ldots$ etc., respectively. Thus, we have for instance the following type annotations for the *SKI* combinators:

$$\begin{array}{lcl}
I & : & (*\alpha \to *\alpha) = (\lambda x : *\alpha.\, x : *\alpha) : (*\alpha \to *\alpha) \\
K & : & (*\alpha \to *\beta \to *\alpha) = (\lambda x : *\alpha.\, \lambda y : *\beta.\, x : *\alpha) : (*\alpha \to *\beta \to *\alpha) \\
S & : & ((*\alpha \to *\beta \to *\gamma) \to (*\alpha \to *\beta) \to *\alpha \to *\gamma) = \\
 & & (\lambda x : (*\alpha \to *\beta \to *\gamma).\, \lambda y : (*\alpha \to *\beta).\, \lambda z : *\alpha.\, ((x\ z)\ (y\ z)) : \\
 & & \quad ((*\alpha \to *\beta \to *\gamma) \to (*\alpha \to *\beta) \to *\alpha \to *\gamma)\ .
\end{array}$$

Polymorphic type schemata require that all occurrences of the same schema identifier be instantiated with the same type.

This polymorphic type concept, which in reference to their inventors is referred to as the Hindley/Milner type system [Hin69, Miln78], has become a standard for many functional (or function-based) languages, e.g.,MIRANDA, SML or HASKELL.

When introducing type annotations (or declarations) into HLFL , they typically have the form

- $expr : \alpha$
  for expressions in general;
- $f(x\_1 : \alpha_1, \ldots, x\_n : \alpha_n) : \beta = expr$ or $f(x\_1 : \alpha_1, \ldots, x\_n : \alpha_n) = expr : \beta$
  for recursive function definitions;
- SUB $(x\_1 : \alpha_1, \ldots, x\_n : \alpha_n)$ IN $expr : \beta$
  for non-recursive function definitions;
- $x : \alpha = expr$ or $x = expr : \alpha$
  for local variable definitions;

- IF $pred\_expr : \alpha_p$ THEN $true\_expr : \alpha_t$ ELSE $false\_expr : \alpha_f$
  for IF-THEN-ELSE clauses with the union type $(\alpha_t \cup \alpha_f)$ and with
  $\alpha_p = \{\text{TRUE}, \text{FALSE}\}$.[14]

- $< expr\_1 : \alpha_1, \ldots, expr\_n : \alpha_n >$ or $< expr\_1, \ldots, expr\_n >:< \alpha_1, \ldots, \alpha_n >$
  for lists with individual component types and
  $< expr\_1, \ldots, expr\_n >: list\_of\ \alpha$
  for lists with uniformly typed components (or well-typed lists).

All type identifiers other than the type α_p for the predicate of an IF-THEN-ELSE clause may be replaced with polymorphic type identifiers.

The atomic type identifiers typically include the basic types *dec_number, int, boolean, char_string*, etc.

Other type identifiers may be defined as

$$\text{TYPE}\ \alpha = type\_expr,$$

where *type_expr* typically specifies an interval of numbers, a set of elements of an enumerative type such as the colors of the rainbow or the days of the week, but also records, union or function types etc.

The scopes of type annotations coincide with binding scopes: type annotations to

- λ-bound variables hold within the respective abstraction bodies;

- expressions hold for the expressions.

Type definitions of the form TYPE $\alpha = type\_expr$ may appear anywhere within the scopes in which they are supposed to be valid.

The rules for consistent type annotations for the terms of a formal calculus (or for programs of a concrete language) are defined by a type system. These rules specify the realization of a type checker (or of a type inference algorithm)peyt87a.

In a static type system, all type checking is done before executing a term (program). Dynamic type systems do the type checking at run-time.

A term (program) is said to be safe (or well-typed) if no type inconsistencies (errors) can occur at run-time. A type system is sound if every term (program) that is typed according to its rules is also safe. Sound type systems allow for static type checking. It essentially ensures that all primitive operators are applied to operands with compatible types. However, it can not statically detect type errors, say, due to division by zero, overflow conditions etc. which are of a dynamic nature and therefore must be dealt with at run-time.

[14]Well-typed expressions require that $\alpha_t = \alpha_f$ in conditionals.

A language (calculus) is said to be strongly typed if the type checker accepts only well-typed programs (terms). Legitimate programs of weakly typed languages may produce type incompatibilities at run-time.

It should be noted here that typing, though widely claimed to introduce more confidence into the correctness of programs, may also cause some rather unpleasant problems which do not occur in untyped languages. Other than outlawing self-applications, the respective type checkers also fail to infer types for a variety of higher-order functions and partial applications.

For instance, a polymorphic type checker infers for the function

$$twice = \text{SUB}(f, x) \text{ IN } f(f(x))$$

the type

$$twice : (*\alpha \to *\alpha) \to *\alpha \to *\alpha \ ,$$

i.e., the function substituted for f has to be of the type $*\alpha \to *\alpha$ and the argument substituted for x has to be of the type $*\alpha$ in order to produce a result of type $*\alpha$. This type perfectly unifies with

- the type of
$$square : int \to int = \text{SUB}(x) \text{ IN } (x * x)$$
and the type int for the value 2 in the application
$$twice(square, 2) \ ,$$
instantiating $*\alpha$ with int,

- the type of
$$\text{REST} : list\_of \ * \beta \to list\_of \ * \beta$$
and the type $list\_of \ \alpha$ for the list $< a\_1, \ldots, a\_n >$ in the application
$$twice(\text{REST}, < a\_1, \ldots, a\_n >) \ ,$$
instantiating $*\alpha$ with $list\_of \ \alpha$.

However, the application

$$twice(\text{FIRST}, < a\_1, \ldots, a\_n >) \ ,$$

causes a type incompatibility since the type $list\_of \ * \beta \to *\beta$ cannot be unified with $*\alpha \to *\alpha$.

Similar typing problems may arise with applications of the same function identifier in different contexts. Consider as an example the combinator

$$S' = \lambda f.\lambda x.\lambda y.(f\ x\ (f\ x\ y))$$

for which a polymorphic type can be inferred as

$$(*\alpha \rightarrow *\beta \rightarrow *\beta) \rightarrow *\alpha \rightarrow *\beta \rightarrow *\beta\ ,$$

i.e., f must be of type $(*\alpha \rightarrow *\beta \rightarrow *\beta)$, x must be of type $*\alpha$ and y must be of type $*\beta$ for a result of type $*\beta$.

However, if this combinator is modified to

$$S'' = \lambda f.\lambda x.\lambda y.(f\ x\ (f\ y))\ ,$$

then the type checker fails for it tries unsuccessfully to infer for the first occurrence of f a type $*\alpha \rightarrow *\beta \rightarrow *\gamma$ and for the second occurrence of f a type $*\alpha \rightarrow *\beta$, and to unify both types (or more precisely the types $*\beta$ and $*\beta \rightarrow *\gamma$). The problem here obviously is that f is considered both a binary and a unary function in S'', whereas in S' it is taken as a binary function in both occurrences.

For a programmer not too familiar with the intricacies of polymorphic type systems these type errors are hardly comprehensible, particularly when occurring in a large program. The standard error messages produced in the above cases are usually of the form

type clash in : *expr*
looking for an : $*\alpha$
I have found an : $*\alpha \rightarrow *\beta$ (or *list_of* $*\ \alpha \rightarrow *\alpha$),

meaning that the type checker failed to unify in the expression *expr* (which may be of considerable size) the types $*\alpha$ and $*\alpha \rightarrow *\beta$ (or *list_of* $*\ \alpha \rightarrow *\alpha$).

5.3.8 The *SECD* Machine

Landin [Lan64] introduced an abstract machine which has proved to be a suitable basic model for concrete evaluators of λ-expressions and, in more or less modified form, for expressions (terms) of concrete programming languages as well [Lan65, Lan66b]. This so-called *SECD* machine derives its name from the fact that it supports, as a run-time environment, four list structures for the representation and manipulation of λ-expressions or of components thereof. These are

- a control structure C which, in the form of a list whose components are λ-terms, holds what may be considered the executable code (or the program text);