λ-Calculus

Christian Sternagel

January 29, 2011

Already Leibniz wanted to *create a universal language in which all possible problems can be stated*. Outcomes of such efforts you may already know are for instance Turing machines and register machines. From there, also the expression *Turing completeness* stems. A computational model is said to be Turing complete if it can compute all effectively computable functions. In this chapter one of those Turing complete formal frameworks is introduced: the λ-calculus (speak "lambda calculus").

1 Syntax

The basic building blocks of the λ-calculus are λ-terms (sometimes called λ-expressions) where the possible shapes of a λ-term t—given a set of variables $\mathcal{V}$—are defined as follows:

$$t \stackrel{\text{def}}{=} x \mid (\lambda x.\, t) \mid (t\ t)$$

Here x is a variable from $\mathcal{V}$, $(\lambda x.\, t)$ is a *(lambda) abstraction* (somehow equivalent to a function definition like $f(x) = t$), and $(t\ t)$ is the *(function) application* of the left λ-term to the right one. The set of all λ-terms that can be built over some set of variables $\mathcal{V}$, is denoted by $\mathcal{T}(\mathcal{V})$.

Haskell' equivalent to abstractions are anonymous functions. The term $(\lambda x.\, x)$ for example is equivalent (modulo typing) to the Haskell function `(\x -> x)` (the backslash should give the optical impression of a lambda).

Example 1.1. Some examples of well-formed λ-terms are:

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

In order to save parentheses the conventions that outermost parentheses are dropped and that applications associate to the left are used. Then the above can be written as

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

Furthermore nested abstractions associate to the right and in order to save λs, variables are grouped together, for example, the rather longish term

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

is written

$$\lambda xyz.\, x\ y\ z$$

using all of the above conventions. (In Haskell `\x y z -> x y z` may be written instead of `(\x -> (\y -> (\z -> ((x y) z))))`.)

1.1 Subterms

The simplest λ-term is a variable. All other λ-terms are built using smaller λ-terms.

Example 1.2. The term $\lambda x.\, x\ x$ consists of an abstraction over the smaller term $x\ x$, whereas this smaller term consists of the application of the term x to the term x.

In the above example, the terms x and $x\ x$ are called *(proper) subterms* of the term $\lambda x.\, x\ x$. The set $\mathcal{S}\mathsf{ub}(t)$ of all subterms of a λ-term t is defined by

$$\mathcal{S}\mathsf{ub}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \{t\} \cup \mathcal{S}\mathsf{ub}(u) & \text{if } t = \lambda x.\, u \\ \{t\} \cup \mathcal{S}\mathsf{ub}(u) \cup \mathcal{S}\mathsf{ub}(v) & \text{if } t = u\ v \end{cases}$$

Note that this also includes the term t itself. A term s is called a *proper* subterm of a term t, if $s \in \mathcal{S}\mathsf{ub}(t)$ and in addition $s \neq t$.

1.2 Free and Bound Variables

The set $\mathcal{V}\mathsf{ar}(t)$ of *variables* occurring in a λ-term t is defined by

$$\mathcal{V}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \{x\} \cup \mathcal{V}\mathsf{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{V}\mathsf{ar}(u) \cup \mathcal{V}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

The set $\mathcal{FV}\mathsf{ar}(t)$ of *free variables* of a term t consists of all variables that occur outside of a lambda abstraction.

$$\mathcal{FV}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \mathcal{FV}\mathsf{ar}(u) \setminus \{x\} & \text{if } t = \lambda x.\, u \\ \mathcal{FV}\mathsf{ar}(u) \cup \mathcal{FV}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

The set $\mathcal{BV}\mathsf{ar}(t)$ of *bound variables* in a term t is defined by

$$\mathcal{BV}\mathsf{ar}(t) \stackrel{\mathsf{def}}{=} \begin{cases} \varnothing & \text{if } t \text{ is a variable} \\ \{x\} \cup \mathcal{BV}\mathsf{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{BV}\mathsf{ar}(u) \cup \mathcal{BV}\mathsf{ar}(v) & \text{if } t = u\ v \end{cases}$$

Definition 1.1 (Closed terms). A λ-term t not having any free variables (that is, $\mathcal{FV}\mathsf{ar}(t) = \varnothing$) is called *closed*.

2 Evaluation of Lambda Expressions

Until now you know how the syntax of the λ-calculus looks like, but it only starts to get interesting after knowing how to do computations using such syntactic constructs. The surprising fact is that the λ-calculus does only need one rule to receive its full computational power. The name of the rule is β.

2.1 Substitutions

Before the β-rule is given, something about *substitutions* has to be said. For the purpose of β-reduction (see the next section), a substitution is a mapping from a variable to a λ-term. We use the notation $\{x/t\}$ to denote the substitution replacing the variable x by the term t.

The application of a substitution $\{x/s\}$ to a λ-term t (written as $t\{x/s\}$) is defined by

$$t\{x/s\} \stackrel{\mathsf{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \neq x \\ (u\{x/s\})\ (v\{x/s\}) & \text{if } t = u\ v \\ t & \text{if } t = \lambda x.\, u \\ \lambda y.\, u\{x/s\} & \text{if } t = \lambda y.\, u,\ y \neq x, \text{ and } y \notin \mathcal{FV}\mathsf{ar}(s) \\ \lambda z.\, u\{y/z\}\{x/s\} & \text{if } t = \lambda y.\, u,\ y \neq x, \text{ and } y \in \mathcal{FV}\mathsf{ar}(s) \end{cases}$$

Note that the variable z in the last case is assumed to be *fresh*, that is, it is different from all the variables occurring in u and s, and also unequal to x. Due to case four, bound variables are not substituted. To understand the last two cases, note what would

happen if it was allowed to apply a substitution $\{x/s\}$ directly to a term $t = \lambda y.\, u$ with $y \in \mathcal{FV}\mathsf{ar}(s)$. This would for example yield $(\lambda x.\, y)\{y/x\} = \lambda x.\, x$ and $(\lambda z.\, y)\{y/x\} = \lambda z.\, x$. Where $\lambda x.\, y$ and $\lambda z.\, y$ provide the same results for the same inputs, but the two λ-terms after the substitution do not behave identical on identical input. (This problem is sometimes referred to as *variable capture*.)

2.2 The β-Rule

Computations within the λ-calculus are done by applying the β-rule stepwise (which is called β-reduction). A special kind of terms are contexts. A context C is built according to the following grammar

$$C \stackrel{\text{def}}{=} \Box \mid \lambda x.\, C \mid t\ C \mid C\ t$$

where $x \in \mathcal{V}$, $t \in \mathcal{T}(\mathcal{V})$, and $\Box$ is a special symbol called *hole*. The set of all contexts built over some set of variables $\mathcal{V}$ is denoted by $\mathcal{C}(\mathcal{V})$. With $C[s]$ we denote the replacement of $\Box$ (in C) by the term s. Note that since every context contains exactly one hole (proving that is left as an exercise) the result of this operation is a term.

The β-rule is defined by

$$s \to_\beta t$$

if there is a subterm u of s (that is, $u \in \mathcal{S}\mathsf{ub}(s)$) that is of the form $u = (\lambda x.\, v)\ w$. Then t is obtained from s by replacing u with $v\{x/w\}$. It is said that s *β-reduces* to t in one step. An alternative way to write this down would be: If there exist a context C and terms s, v, and w, such that

$$s = C[(\lambda x.\, v)\ w]$$

(which is equivalent to stating that $(\lambda x.\, v)\ w$ is a subterm of s) then

$$s \to_\beta C[v\{x/w\}]$$

Consider for example the reduction step

$$(\lambda x.\, x)\ (\lambda x.\, x) \to_\beta x\{x/\lambda x.\, x\} = \lambda x.\, x$$

Here $(\lambda x.\, x)\ (\lambda x.\, x)$ is called a *redex* (the short form of *red*ucible *ex*pression) and $\lambda x.\, x$ is its *contractum*. In principle this alone is sufficient to define and evaluate every effectively computable function. The only remaining question is: What is the result of a computation?

2.3 Normal Forms

A λ-term is said to be in *normal form* (NF) if it is not possible to apply any β-reduction. A normal form can be considered as the outcome of a computation. Note that there are λ-terms that do not have any normal form. For others it might depend on the order of β-steps whether a normal form is reached or not.

Example 2.1. Reducing the term $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ results in the following reduction sequence

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta \ldots$$

In fact this term does not have a normal form. Compare the above term with the following $(\lambda yz.\, z)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))$. There are two redexes: The first one is the whole term itself and the second one is $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. If the first one is contracted then a normal form is reached immediately (namely $\lambda z.\, z$), but the second redex can be contracted indefinitely.

3 Representing Data Types in the λ-Calculus

To get a grasp of the power of the λ-calculus it is shown how some data types and operations on them that are frequently used in functional programming languages—like Booleans with Boolean connectives, integers with integer arithmetic, pairs, and lists with list operations—can be encoded in the λ-calculus.

3.1 Booleans and Conditionals

Consider an expression like **`if`** b **`then`** t **`else`** e. To encode this as a λ-term something of the shape $\lambda bte.\, s$ is needed, where s has to specify that if b holds then the result should be t and otherwise it should be e. In order to achieve such behavior of s the Boolean values `true` and `false` have to be encoded as λ-terms. One nice possibility is

$$\mathsf{True} \stackrel{\mathsf{def}}{=} \lambda xy.\, x$$
$$\mathsf{False} \stackrel{\mathsf{def}}{=} \lambda xy.\, y$$

Then the **`if`** b **`then`** t **`else`** e of Haskell can be encoded as

$$\mathsf{if} \stackrel{\mathsf{def}}{=} \lambda xyz.\, x\ y\ z$$

since

$$\mathsf{if}\ \mathsf{True}\ t\ e = (\lambda xyz.\, x\ y\ z)\ (\lambda xy.\, x)\ t\ e \to_\beta^3 (\lambda xy.\, x)\ t\ e \to_\beta^2 t$$

and

$$\mathsf{if}\ \mathsf{False}\ t\ e = (\lambda xyz.\, x\ y\ z)\ (\lambda xy.\, y)\ t\ e \to_\beta^3 (\lambda xy.\, y)\ t\ e \to_\beta^2 e$$

3.2 Natural Numbers

One way to encode (natural) numbers, that is, only the non-negative part of integers, in the λ-calculus are the so called *Church numerals*.

Definition 3.1. Let s and t be λ-terms, and n be a natural number (that is, $n \in \mathbb{N}$[1]). Then $s^n\ t$ is defined inductively by

$$s^0\ t \stackrel{\text{def}}{=} t$$
$$s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

The Church numerals (represented by $\bar{0}$, $\bar{1}$, $\bar{2}$, ... in the following) are defined by

$$\bar{\mathsf{n}} \stackrel{\text{def}}{=} \lambda f x.\, f^n\ x$$

Example 3.1. Using the above definition the first four Church numerals are given by

$$\bar{0} \stackrel{\text{def}}{=} \lambda f x.\, x$$
$$\bar{1} \stackrel{\text{def}}{=} \lambda f x.\, f\ x$$
$$\bar{2} \stackrel{\text{def}}{=} \lambda f x.\, f\ (f\ x)$$
$$\bar{3} \stackrel{\text{def}}{=} \lambda f x.\, f\ (f\ (f\ x))$$

On first sight this definition does not look very obvious (a reason for that could be that it is not), however using the above encoding for numbers, the definitions of addition, multiplication, and exponentiation are very short, namely

$$\mathsf{add} \stackrel{\text{def}}{=} \lambda m n f x.\, m\ f\ (n\ f\ x)$$
$$\mathsf{mult} \stackrel{\text{def}}{=} \lambda m n f.\, m\ (n\ f)$$
$$\mathsf{exp} \stackrel{\text{def}}{=} \lambda m n.\, n\ m$$

3.3 Pairs

Concerning pairs, some means to construct them—given two values—and to select the first and second component respectively, are needed. This is done via the λ-terms

$$\mathsf{pair} \stackrel{\text{def}}{=} \lambda x y f.\, f\ x\ y$$
$$\mathsf{fst} \stackrel{\text{def}}{=} \lambda p.\, p\ \mathsf{True}$$
$$\mathsf{snd} \stackrel{\text{def}}{=} \lambda p.\, p\ \mathsf{False}$$

[1] For the purpose of this lecture $\mathbb{N}$ does always denote the set $\{0, 1, 2, 3, \dots\}$ of positive integers together with 0 (which itself is neither negative nor positive).

The reader may already have missed subtraction on natural numbers. The reason is, that pairs are needed before subtraction can be defined. For Church numerals subtraction is an awfully complex (and slow) operation. The first problem is how to get $\lambda fx.\, f^n\ x$ from $\lambda fx.\, f^{n+1}\ x$, that is, the predecessor function. Consider the function

$$\mathsf{pairsucc} \stackrel{\mathsf{def}}{=} \lambda fp.\, \mathsf{pair}\ (f\ (\mathsf{fst}\ p))\ (\mathsf{fst}\ p)$$

which may be easier understandable using pattern matching, that is,

$$\mathsf{pairsucc} \stackrel{\mathsf{def}}{=} \lambda f(x, \_).\, (f\ x, x)$$

The idea is, to pair the first component of a given pair with its successor (where for Church numerals, computing the successor means to add an additional f).

If $(\mathsf{pairsucc}\ f)$ is applied $n{+}1$ times to an argument pair (x, y) then the result is obviously $(\mathsf{pairsucc}\ f)^{n+1}\ (x, y) = (f^{n+1}\ x, f^n\ x)$. The encoding of a Church numeral $\lambda fx.\, f^n\ x$ basically is the function that applies f^n to x (that is, f is applied n times to x). Hence the result of

$$(\lambda fx.\, f^{n+1}\ x)\ (\mathsf{pairsucc}\ f)\ (\mathsf{pair}\ x\ x)$$

is $(f^{n+1}\ x, f^n\ x)$ and by selecting the second component the predecessor $\overline{\mathsf{n}}$ of $\overline{\mathsf{n{+}1}}$ can be obtained. This facilitates the definitions

$$\begin{aligned}\mathsf{pre} &\stackrel{\mathsf{def}}{=} \lambda nfx.\, \mathsf{snd}\ (n\ (\mathsf{pairsucc}\ f)\ (\mathsf{pair}\ x\ x))\\ \mathsf{sub} &\stackrel{\mathsf{def}}{=} \lambda mn.\, n\ \mathsf{pre}\ m\end{aligned}$$

for the predecessor function and subtraction $(m - n)$. Note that by this definition we have $\mathsf{pre}\ \overline{\mathsf{0}} \to_\beta \overline{\mathsf{0}}$ and thus, if $\overline{\mathsf{n}}$ is larger than $\overline{\mathsf{m}}$, we obtain $\mathsf{sub}\ \bar{m}\ \bar{n} \to_\beta^* \overline{\mathsf{0}}$.

3.4 Lists

Having pairing and Booleans, a nonempty list $x : y$ can be encoded by the nested pairs $\mathsf{pair}\ \mathsf{False}\ (\mathsf{pair}\ x\ y)$ (or $(\mathsf{False}, (x, y))$, if we use syntactic sugar for pairs) where False denotes that the list is not empty. Before defining the encoding for empty lists, consider the functions that should work on lists. Those are: head, tail, null (checking whether a list is empty), and cons. Most of them are easy:

$$\begin{aligned}\mathsf{cons} &\stackrel{\mathsf{def}}{=} \lambda xy.\, \mathsf{pair}\ \mathsf{False}\ (\mathsf{pair}\ x\ y)\\ \mathsf{head} &\stackrel{\mathsf{def}}{=} \lambda z.\, \mathsf{fst}\ (\mathsf{snd}\ z)\\ \mathsf{tail} &\stackrel{\mathsf{def}}{=} \lambda z.\, \mathsf{snd}\ (\mathsf{snd}\ z)\end{aligned}$$

Now for null the desired result for empty lists is True whereas for nonempty lists it is False. For nonempty lists it would suffice to return the first component of the given pair.

It only remains to define nil in a way that $\mathsf{fst}\ l$ evaluates to False for nonempty lists and to True for empty lists. Since $\mathsf{fst} = \lambda p.\, p\ \mathsf{True}$ the solution is

$$\mathsf{nil} \stackrel{\mathsf{def}}{=} \lambda x.\, x$$
$$\mathsf{null} \stackrel{\mathsf{def}}{=} \mathsf{fst}$$

4 Recursion

Consider an implementation of a recursive function in the λ-calculus. For instance the list function `length`. In Haskell it could be implemented by

```
length x = if null x then 0
                     else 1 + length (tail x)
```

Hence it should be possible to write something like

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \lambda x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (\mathsf{length}\ (\mathsf{tail}\ x)))$$

The only problem here is that the definition of length already needs the length function (which after all is the point of recursive definitions). The idea is to extend length by an additional parameter f and replace all occurrences of length within its definition by this parameter. The result is

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (f\ (\mathsf{tail}\ x)))$$

Since in the end, length should only take one argument, another λ-term has to be added. Currently it is not clear how this term should look like, but lets call it Y. Then length is defined by

$$\mathsf{length} \stackrel{\mathsf{def}}{=} \mathsf{Y}\ (\lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (f\ (\mathsf{tail}\ x))))$$

Suppose that Y has the property that for every λ-term t it holds that $\mathsf{Y}\ t \equiv t\ (\mathsf{Y}\ t)$[2] then the following reduction is possible (where length corresponds to $\mathsf{Y}\ t$):

$$\begin{aligned}\mathsf{length} &\to_\beta^* (\lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (f\ (\mathsf{tail}\ x))))\ \mathsf{length}\\ &\to_\beta \lambda x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (\mathsf{length}\ (\mathsf{tail}\ x)))\end{aligned}$$

which yields the desired result. Indeed such a Y exists.

[2] Here $\equiv$ means that the results of the computations on both sides are the same. Formally $s \equiv t$ amounts to $s \leftrightarrow_\beta t$, that is, applying β-steps in arbitrary directions for an arbitrary number of times. Or put differently, the symmetric, transitive, and reflexive closure of $\to_\beta$.

Definition 4.1. The *'Y'-combinator* Y—discovered by Haskell B. Curry—is defined by

$$\mathsf{Y} \stackrel{\text{def}}{=} \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

and has the *fixed point*[3] *property*, that is, for all λ-terms t

$$\mathsf{Y}\ t \equiv t\ (\mathsf{Y}\ t)$$

A remarkable result is that the content of this chapter until here is sufficient to encode almost all Haskell programs that were implemented so far during the lecture, only using the λ-calculus. For example, strings are lists of characters, however, on a computer characters are essentially numbers.

5 Evaluation Strategies

The *evaluation strategy* (or *strategy* for short) determines which redex to choose if there is more than one possibility. Until now the decision was arbitrary, but when implementing β-reduction the computer needs to know exactly what to do. Two "natural" choices of evaluation strategies are outlined in the following.

5.1 Outermost Reduction

The (leftmost) outermost strategy always chooses the (leftmost) outermost redex in a term to apply a β-step. An outermost redex is one that is not a subterm of some other redex.

5.2 Innermost Reduction

The (leftmost) innermost strategy always chooses the (leftmost) innermost redex in a term to apply a β-step. An innermost redex is one that does not contain any redexes as proper subterms.

[3] Intuitively speaking, a *fixed point* of a function f is a value v such that applying f to v always results in v. Consequently $v = f\ v = f\ (f\ v) = \cdots = f^n\ v$ for an arbitrary $n \in \mathbb{N}$. The fixed point combinator somehow computes such a fixed point for a given function.

5.3 Call-by-Value vs. Call-by-Name

From the above (rewrite) strategies two evaluation strategies for functional programs can be extracted. The first is called *call-by-value* and the second *call-by-name*. Call-by-value is the evaluation strategy adopted by most programming languages. In this evaluation strategy the arguments of a function are evaluated before the function is called on them. Since Haskell is a *lazy* language, it adopts call-by-name, which means that function arguments are only evaluated on demand. For example the function call

```
let f x = x + 1 in f (3 + 2)
```

will be evaluated in the following order in Haskell

$$\begin{aligned} \texttt{f (3 + 2)} &= \texttt{(3 + 2) + 1} \\ &= \texttt{5 + 1} \\ &= \texttt{6} \end{aligned}$$

Still, it is thinkable to do the derivation in a different order, as in

$$\begin{aligned} \texttt{f (3 + 2)} &= \texttt{f 5} \\ &= \texttt{5 + 1} \\ &= \texttt{6} \end{aligned}$$

which would be call-by-value, that is, first evaluate arguments and then functions. It can be seen that there is a tight correspondence between call-by-name and outermost reduction as well as between call-by-value and innermost reduction (so far there is no difference at all; however, there actually is a slight difference as will be seen next).

In addition to only reduce outermost (innermost) redexes, call-by-name (call-by-value) does only consider terms that are in *weak head normal form*.

Definition 5.1. A λ-term t is said to be in *weak head normal form* (WHNF) if it is not a function application, that is, there do not exist λ-terms u and v such that

$$t = u\ v$$

Intuitively this means that functions without arguments are considered as normal forms, for example, in

```
foo = (\x -> 1 + 2)
```

`foo` is not reduced to `\x -> 3` as long as it does not get any argument. In other words call-by-name corresponds to outermost reduction to WHNF whereas call-by-value corresponds to innermost reduction to WHNF.

Example 5.1. Consider the λ-term $\mathsf{length}\ \mathsf{nil}$, computing the length of the empty list. This corresponds to the term

$$(\mathsf{Y}\ (\lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (f\ (\mathsf{tail}\ x)))))\ (\lambda x.\, x)$$

having the eight redexes

$$\mathsf{tail}\ x \tag{1}$$
$$\mathsf{add}\ \bar{1} \tag{2}$$
$$\mathsf{null}\ x \tag{3}$$
$$\mathsf{snd}\ x \tag{4}$$
$$\mathsf{if}\ (\mathsf{null}\ x) \tag{5}$$
$$\mathsf{snd}\ (\mathsf{snd}\ z) \tag{6}$$
$$(\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)) \tag{7}$$
$$\mathsf{Y}\ (\lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ \bar{0}\ (\mathsf{add}\ \bar{1}\ (f\ (\mathsf{tail}\ x)))) \tag{8}$$

where (4) and (6) are hidden in the definition of tail and (7) is hidden in the definition of Y. If scanning for redexes starting at the left, then the first one obtained is (8), which turns out to be the leftmost outermost redex since it is not a subterm of any other redex. Using an outermost strategy will yield the result in some reduction steps. However, the leftmost innermost redex of the above term that is not in WHNF is (7). This term is the starting point of the reduction sequence

$$\begin{aligned}(\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)) &\to_\beta f\ ((\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)))\\ &\to_\beta f\ (f\ ((\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))))\\ &\ldots\end{aligned}$$

using innermost reduction to WHNF. This does not terminate. Indeed every recursive definition using Y is nonterminating under call-by-value evaluation.

As can be seen from the above example, Y is not suitable for call-by-value reduction. Gladly there is an alternative to Y which does work also in this case.

Definition 5.2. The *'Z'-combinator* Z—which is a slight variation of Y—is a fixed point combinator (that is, having the fixed point property) that can be used for call-by-value reduction and is defined by

$$\mathsf{Z} \stackrel{\mathsf{def}}{=} \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y))$$

As already mentioned, two different evaluation strategies are considered. Call-by-value is tightly connected to *strict* or *eager* evaluation (that is adopted by most imperative programming languages). A similar connection can be found between call-by-name and *non-strict* or *lazy*[4] evaluation (as adopted by Haskell).

[4]Lazy evaluation and call-by-name are not the same, but they are quite similar. Lazy evaluation corresponds to call-by-name evaluation where additionally a technique called sharing (or memoization) is used to avoid multiple computations of the same expression. However, that's a different story.

6 Chapter Notes

Similar examples and further information can be found in [1, 2, 3].

References

[1] Henk P. Barendregt and Erik Barendsen. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[2] Anthony J. Field and Peter Harrison. *Functional Programming*. Addison-Wesley, 1988.

[3] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.