λ-Calculus

Christian Sternagel

November 9, 2017

A little bit of syntactic sugar helps you swallow the lambda calculus.

Peter Landin

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 [7] 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") [2].

1 Syntax

The basic building blocks of the λ-calculus are λ-terms (or λ-expressions).

Definition 1 (Raw terms). The possible shapes of a *raw λ-term* t are defined by the following BNF grammar:

$$t ::= x \mid (t\ t) \mid (\lambda x.\, t)$$

Here x is a *variable (or atom)*, $(t\ u)$ is the *application* of the left λ-term to the right one (the idea is that t encodes a function that can be applied to input u), and $(\lambda x.\, t)$ is a *(lambda) abstraction* (somewhat similar to a function definition like $f(x) = t$).

The set of all *raw λ-terms* is denoted by $\mathfrak{R}$ (deliberately using an odd symbol, so that we will try hard to avoid it).

Remark. The 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 visual impression of a lambda).

Example 1. Some examples of raw 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 (1) outermost parentheses are dropped, (2) application binds tighter than abstraction, and (3) 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 compressed to

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

using the above conventions. (The same conventions apply also to Haskell, where for example `\x y z -> x y z` is the short form of `(\x -> (\y -> (\z -> ((x y) z))))`.)

1.1 α-Equivalence

As in programming, the names of parameters are not supposed to change the meaning of a function. For example, `id x = x` is the same function as `id y = y`. To make this irrelevance of "names" more formal the notion of *α-equivalence* is introduced.

But before we can do so, we need to be able to talk about renaming of variables and the set of free variables of a term.

Definition 2 (Renaming variables). Variables are renamed by applying permutations π (between variables) to terms as follows:[1]

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

[1] A permutation is a bijective function π such that $\pi(x) \neq x$ for only finitely many variables x; it can be equivalently expressed by a finite list of *swappings*.

Thus, the permutation π is applied to every variable occurrence throughout a given term.

Example 2. Consider the permutation $(x \rightleftharpoons y)$, replacing xs by ys and vice versa. Then

$$(x \rightleftharpoons y) \bullet (\lambda xy.\, x\ (\lambda zx.\, z\ x)) = \lambda yx.\, y\ (\lambda zy.\, z\ y)$$

Definition 3 (Free variables). The set of *free variables* $\mathcal{F}(t)$ of a (raw) λ-term t consists of all variables that occur outside the scope of a lambda abstraction:

$$\begin{aligned} \mathcal{F}(x) &= \{x\} \\ \mathcal{F}(t\ u) &= \mathcal{F}(t) \cup \mathcal{F}(u) \\ \mathcal{F}(\lambda x.\, t) &= \mathcal{F}(t) \setminus \{x\} \end{aligned}$$

(Raw) terms without free variables (that is, $\mathcal{F}(t) = \varnothing$) are called *closed*.

Definition 4 (Freshness constraints). We say that a set of variables X is *fresh for* a given syntactic object o—written $X \sharp o$—if none of the variables in X occurs in the "free variables" of o. For terms the "free variables" are given by $\mathcal{F}$ and for permutations by their domain (that is, the set of variables that are modified by the permutation).

The curious reader might be interested in the following formal definition of α-*equivalence* ($\equiv_\alpha$). If you prefer an informal account, you may skip to the subsequent remark.

Definition 5 (α-Equivalence). The notion of α-*equivalence* is introduced simultaneously with an auxiliary notion of equivalence between variable-term pairs with respect to a given permutation ($\equiv_\pi$). The pair (x, t) is π-*equivalent* to the pair (y, u)—written $(x, t) \equiv_\pi (y, u)$—if and only if all of the following conditions are satisfied

$$\mathcal{F}(t) \setminus \{x\} = \mathcal{F}(u) \setminus \{y\} \qquad \mathcal{F}(t) \setminus \{x\} \sharp \pi \qquad \pi \bullet t \equiv_\alpha u \qquad \pi(x) = y$$

At the same time α-equivalence is defined inductively as follows

$$\frac{}{x \equiv_\alpha x} \qquad \frac{t \equiv_\alpha t' \quad u \equiv_\alpha u'}{t\ u \equiv_\alpha t'\ u'} \qquad \frac{(x, t) \equiv_\pi (y, u) \text{ for some } \pi}{\lambda x.\, t \equiv_\alpha \lambda y.\, u}$$

Remark. While the above is a formally correct definition of α-equivalence, it might be instructive to just remember that two raw terms are α-equivalent if one is obtained from the other by consistently renaming its bound variables (which might require to apply several renaming permutations in a row).

As the name suggests, α-equivalence is an equivalence relation on raw terms. Thus, we are finally in a position to obtain λ-terms (notice the lack of the prefix "raw").

Definition 6 (λ-Terms). The set of λ-terms Λ is obtained by taking the quotient of the set of raw terms with respect to α-equivalence. Formally: $\Lambda = \mathfrak{R} / \equiv_\alpha$.

Mathematically, a single term is represented by a whole set of raw terms: its α-*equivalence class*. Thus, for example, when writing $\lambda xy.\, y$; we really mean

$$\{\ldots, \lambda xy.\, y, \ldots, \lambda yx.\, x, \ldots, \lambda ab.\, b, \ldots\}$$

With this definition of λ-terms as a quotient, α-equivalent terms are in fact equal (because their α-equivalence classes coincide)!

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 a single rule to receive its full computational power. Before this rule (called the "β-rule," for historical reasons) can be given, we need to talk about *substitutions* though.

2.1 Substitutions

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

$$\begin{aligned}
x[x := s] &= s \\
y[x := s] &= y && \text{if } x \neq y \\
(t\ u)[x := s] &= (t[x := s])\ (u[x := s]) \\
(\lambda y.\, t)[x := s] &= \lambda y.\, (t[x := s]) && \text{if } y \sharp (x, s)
\end{aligned}$$

Note that the variable y in the last case is assumed to be *fresh for* x and s, that is, it is different from all the free variables of s and also unequal to x. Thus, bound variables are not substituted.

As an indication for why the freshness condition in the last case is required, note what would happen if it was allowed to apply a substitution $[x := s]$ directly to a term $t = \lambda y.\, u$ with $y \in \mathcal{F}(s)$. This would for example yield $(\lambda x.\, y)[y := x] = \lambda x.\, x$ and $(\lambda z.\, y)[y := x] = \lambda z.\, x$. While the α-equivalent raw λ-terms $\lambda x.\, y$ and $\lambda z.\, y$ provide the same results for the same inputs (and actually represent one and the same λ-term), the two (non-α-equivalent) raw λ-terms $\lambda x.\, x$ (resulting from a wrong substitution) and $\lambda z.\, x$ do not behave identical (and indeed represent two different λ-terms). (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).

Definition 7 (β-Reduction). The β-rule is defined inductively by

$$\frac{x \sharp u}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)} \qquad \frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)}$$

So if there is a subterm u of s that is of the form $u = (\lambda x.\, v)\ w$ where x is fresh for w, then $s \to_\beta t$, where t is obtained from s by replacing u with $v[x := w]$. We say that s *β-reduces* to t in one step.

Note that it is always possible to satisfy the freshness side condition by appropriately renaming bound variables first (since our λ-terms are actually $\equiv_\alpha$-equivalence classes, doing so does not even change the term).

Let s and t be λ-terms. If s reduces to t in a number of β-steps then we denote this by $s \to_\beta^* t$ (or $s \to_\beta^n t$ to make concrete that the reduction has exactly n steps).

Consider for example the reduction step

$$\underline{((\lambda x.\, x)\ (\lambda x.\, x))}\ y \to_\beta (x[x := \lambda x.\, x])\ y = (\lambda x.\, x)\ y$$

where the underlined expression $(\lambda x.\, x)\ (\lambda x.\, x)$ is called a *redex* (the short form of *red*ucible *ex*pression) and $\lambda x.\, x$ is called its *contractum*. The resulting term $(\lambda x.\, x)\ y$ is called the *reduct* of the step. In principle this alone is sufficient to define and evaluate every effectively computable function. At least, as long as we can answer the question: 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 in which β-steps are applied whether a normal form is reached or not.

Example 3. Reducing the term $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ results in the infinite reduction sequence

$$\underline{(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)} \to_\beta \underline{(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)} \to_\beta \underline{(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)} \to_\beta \ldots$$

In fact, the term $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ does not have a normal form. Compare this with the term $\underline{(\lambda yz.\, z)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))}$. There are two (underlined) redexes: the whole term itself and the subterm $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. If the first one is contracted then the normal form $\lambda z.\, z$ is reached immediately, 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 possibility is

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

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

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

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 8 (Church numerals). Let s and t be λ-terms, and n be a natural number (that is, $n \in \mathbb{N}$[2]). Then $s^n\ t$ is defined inductively by

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

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

$$\mathsf{n} = \lambda fx.\, f^n\ x$$

[2]For the purpose of this note $\mathbb{N}$ always denotes the set $\{0, 1, 2, 3, \dots\}$ of non-negative integers.

Example 4. Using the above definition, the first four Church numerals are given by

$$\begin{aligned}\mathsf{0} &= \lambda fx.\, x\\ \mathsf{1} &= \lambda fx.\, f\ x\\ \mathsf{2} &= \lambda fx.\, f\ (f\ x)\\ \mathsf{3} &= \lambda fx.\, f\ (f\ (f\ x))\end{aligned}$$

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 of numbers, the definitions of addition, multiplication, and exponentiation are very succinct, namely

$$\begin{aligned}\mathsf{add} &= \lambda mnfx.\, m\ f\ (n\ f\ x)\\ \mathsf{mult} &= \lambda mnf.\, m\ (n\ f)\\ \mathsf{exp} &= \lambda mn.\, n\ m\end{aligned}$$

Example 5. To familiarize ourselves with Church numerals we reduce the λ-term $\mathsf{add\ 1\ 2}$ to normal form (the contracted redex is underlined in each step):

$$\begin{aligned}\mathsf{add\ 1\ 2} &= \underline{(\lambda mnfx.\, m\ f\ (n\ f\ x))\ \mathsf{1}\ \mathsf{2}}\\ &\to_\beta \underline{(\lambda nfx.\, \mathsf{1}\ f\ (n\ f\ x))\ \mathsf{2}}\\ &\to_\beta \lambda fx.\, \mathsf{1}\ f\ (\mathsf{2}\ f\ x)\\ &= \lambda fx.\, \mathsf{1}\ f\ (\underline{(\lambda gy.\, g\ (g\ y))\ f}\ x)\\ &\to_\beta \lambda fx.\, \mathsf{1}\ f\ (\underline{(\lambda y.\, f\ (f\ y))\ x})\\ &\to_\beta \lambda fx.\, \mathsf{1}\ f\ (f\ (f\ x))\\ &= \lambda fx.\, \underline{(\lambda gy.\, g\ y)\ f}\ (f\ (f\ x))\\ &\to_\beta \lambda fx.\, \underline{(\lambda y.\, f\ y)\ (f\ (f\ x))}\\ &\to_\beta \lambda fx.\, f\ (f\ (f\ x)) = \mathsf{3}\end{aligned}$$

After we have seen that $\mathsf{add\ 1\ 2} \to_\beta^* \mathsf{3}$, we investigate why the add combinator works as it should. First consider the $\lambda mnfx$ prefix. Here, the m and n are the two parameters that add expects. The f and the x are needed since the result of $\mathsf{add\ m\ n}$ should be a Church numeral again, that is, of the shape $\lambda fx.\, t$ for some term t. Now what happens if we apply add to Church numerals m and n? We have

$$\mathsf{add\ m\ n} = (\lambda mnfx.\, m\ f\ (n\ f\ x))\ \mathsf{m\ n} \to_\beta^2 (\lambda fx.\, \mathsf{m}\ f\ (\mathsf{n}\ f\ x))$$

Before we continue, look at the redex $\mathsf{n}\ f\ x$. We have

$$\mathsf{n}\ f\ x = (\lambda g\ y.\, g^n\ y)\ f\ x \to_\beta^2 f^n\ x$$

and hence for the redex $\mathsf{m}\ f\ (\mathsf{n}\ f\ x)$ we obtain

$$\mathsf{m}\ f\ (\mathsf{n}\ f\ x) \to_\beta^2 \mathsf{m}\ f\ (f^n\ x) = (\lambda g\ y.\, g^m\ y)\ f\ (f^n\ x) \to_\beta^2 f^m\ (f^n\ \ x) = f^{m+n}\ x$$

The last identity is left as an exercise. In total, we have $\mathsf{add\ m\ n} \to^*_\beta \mathsf{m} + \mathsf{n}$.

The reasoning for mult is similar but a bit more involved. We start from

$$\mathsf{mult\ m\ n} = (\lambda mnf.\, m\ (n\ f))\ \mathsf{m\ n} \to^2_\beta \lambda f.\, \mathsf{m}\ (\mathsf{n}\ f)$$

and for the redex $\mathsf{n}\ f$ we have

$$\mathsf{n}\ f = (\lambda gx.\, g^n\ x)\ f \to_\beta \lambda x.\, f^n\ x$$

and hence for the redex $\mathsf{m}\ (\mathsf{n}\ f)$ we obtain

$$\begin{aligned}\mathsf{m}\ (\mathsf{n}\ f) = (\lambda gx.\, g^m\ x)\ ((\lambda gx.\, g^n\ x)\ f) &\to_\beta (\lambda gx.\, g^m\ x)\ (\lambda x.\, f^n\ x)\\ &\to_\beta \lambda x.\, (\lambda y.\, f^n\ y)^m\ x \to^*_\beta \lambda x.\, f^{nm}\ x\end{aligned}$$

We leave the proof of $(\lambda y.\, f^n\ y)^m\ x \to^*_\beta f^{nm}\ x$ as an exercise. In total, we obtain $\mathsf{mult\ m\ n} \to^*_\beta \mathsf{mn}$.

3.3 Pairs

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

$$\begin{aligned}\mathsf{Pair} &= \lambda xyf.\, f\ x\ y\\ \mathsf{fst} &= \lambda p.\, p\ \mathsf{True}\\ \mathsf{snd} &= \lambda p.\, p\ \mathsf{False}\end{aligned}$$

The vigilant reader may already have missed subtraction on natural numbers. The reason for this omission is that pairs are needed before subtraction can be defined. For Church numerals subtraction is an awfully complex (and costly) operation. We will compute subtraction by repeatedly applying the predecessor function. Hence the first problem is to obtain $\lambda fx.\, f^n\ x$ from $\lambda fx.\, f^{n+1}\ x$, that is, get the Church numeral n from n+1 by computing the predecessor function. Consider the function

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

which may be more readable using pattern matching and syntactic sugar for pairs

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

If $(\mathsf{pairsucc}\ f)$ is applied $n+1$ times to an argument pair (x, y) then the result is $(\mathsf{pairsucc}\ f)^{n+1}\ (x, y) = (f^{n+1}\ x, f^n\ x)$ (as can be proved by induction on n). 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 n of n+1 can be obtained. This facilitates the definitions

$$\begin{aligned}\mathsf{pre} &= \lambda n f x.\, \mathsf{snd}\ (n\ (\mathsf{pairsucc}\ f)\ (\mathsf{Pair}\ x\ x))\\ \mathsf{sub} &= \lambda m n.\, n\ \mathsf{pre}\ m\end{aligned}$$

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

3.4 Lists

Having pairing and Booleans, a nonempty list `x : y` can be encoded by the nested pairs Pair False (Pair $x\ y$) (or (False, (x, y)), if we use syntactic sugar for pairs) where False indicates 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:

$$\begin{aligned}\mathsf{Cons} &= \lambda x y.\, \mathsf{Pair\ False}\ (\mathsf{Pair}\ x\ y)\\ \mathsf{head} &= \lambda z.\, \mathsf{fst}\ (\mathsf{snd}\ z)\\ \mathsf{tail} &= \lambda z.\, \mathsf{snd}\ (\mathsf{snd}\ z)\end{aligned}$$

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 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

$$\begin{aligned}\mathsf{Nil} &= \lambda x.\, x\\ \mathsf{null} &= \mathsf{fst}\end{aligned}$$

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

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

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 from above by an additional parameter f and replace all occurrences of length by this parameter. The result is

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

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

$$\mathsf{length} = \mathsf{Y}\ \mathsf{length}'$$

Suppose that Y has the property that for every λ-term t it holds that $\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t)$ (where $\equiv_\beta$ means that we can apply arbitrarily many β-steps in both directions; formally, $\equiv_\beta$ is the symmetric, transitive, and reflexive closure of $\to_\beta$) then the following reduction is possible (where length corresponds to $\mathsf{Y}\ t$):

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

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

Definition 9. The *'Y'-combinator* Y—discovered by Haskell B. Curry—is defined by

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

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

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t)$$

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 Reduction Strategies

A *reduction 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 reduction strategies are outlined in the following.

[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 $n \in \mathbb{N}$. A fixed point combinator computes such a fixed point for a given function.

Definition 10 (Applicative Order). *Applicative order reduction* chooses the (necessarily unique) rightmost innermost redex first.

Intuitively, applicative order reduction always reduces the arguments of a function before the function call itself. Or put differently: functions are only applied to normal forms.

Definition 11 (Normal Order). *Normal order reduction* chooses the (necessarily unique) leftmost outermost redex first.

The "normal" in normal order reduction, stems from the fact that using this reduction strategy, whenever a normal form exists, reduction will definitely reach a normal form eventually.

Before we can introduce further reduction strategies (which are closer to actual evaluation strategies that are used by programming languages) we need two ways to classify λ-terms.

Definition 12 (Values). A λ-term that is not an application is called a *value*.

Definition 13 (WHNFs). A λ-term is said to be in *weak head normal form* (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Definition 14 (Call by Value). *Call by value reduction* stops at values and otherwise chooses an outermost redex whose right-hand side has already reduced to a value.

Like applicative order reduction, call by value reduction reduces arguments of functions first. But in addition, it never "peeks" below a lambda.

Definition 15 (Call by Name). *Call by name reduction* chooses the same redex as normal order reduction, but stops as soon as a WHNF is reached.

In particular, call by name reduction does not take place below lambdas either.

Call by value reduction and call by name reduction are tightly connected to two evaluation strategies for programming languages of the same name: *call by value evaluation* and *call by name evaluation*. Call by value evaluation is a *strict* or *eager* evaluation strategy that is adopted by most programming languages. *Non-strict* or *lazy* programming languages, like Haskell, adopt variants of call by name evaluation, which usually means that function arguments are only evaluated on demand.[4]

[4] Lazy evaluation and call by name are not the same, but 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 (this combination is one possible implementation of call by need evaluation). However, that's a different story.

Example 6. For example the function call to `f` in

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

will be evaluated in the following order in Haskell

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

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

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

which would be call by value evaluation.

Note that call by name reduction means that the body of a function without arguments is not reduced (since it is in WHNF), for example, in

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

The redex `1 + 2` inside `foo` is not reduced as long as `foo` does not get any argument. This is of practical interest because if the function `foo` is never applied to any argument in the remainder of the program then the (useless) effort of evaluating `1 + 2` is avoided. (Note that in general the body of a function might contain more costly, possibly even non-terminating, computations.)

Example 7. Consider the λ-term $\mathsf{length}\ \mathsf{Nil}$, computing the length of the empty list. This corresponds to the term

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

having the eight redexes

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

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 0 in some reduction steps. However, call by value reduction of the above term causes the reduction

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

which does not terminate. Indeed, every recursive definition using Y is nonterminating under call by value reduction.

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

Definition 16. The *'Z'-combinator* Z (a slight variation of Y) is defined by

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

and satisfies the property $\mathsf{Z}\ t \equiv_\beta t\ (\lambda x.\, \mathsf{Z}\ f\ x)$ with $x \sharp \mathsf{Z}\ f$. While this is not exactly the fixed point property, it is close enough in practice.[5]

But note that with call by value, also other constructs have to be modified (the astute reader might already have noticed that Z length′ Nil still does not terminate). For example, `if ... then ... else` is usually assumed to be evaluated non-strictly. This can be achieved for if by judiciously sprinkling in dummy abstractions and dummy arguments, as in the following variant of length′ which together with Z yields a definition of length that also works with call by value:

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

6 Chapter Notes

Similar examples and further information can be found in the literature [1,4,6]. An important property of the λ-calculus is that β-reduction (that is, *computation*) gives unique results: there is no term s such that $s \to_\beta^* t$ and $s \to_\beta^* u$ for different normal forms t and u. The lambda calculus satisfies this property since it satisfies the stronger "Church-Rosser property," which guarantees that for all terms t and u with $t \equiv_\beta u$ we can find a common reduct v of t and u, that is, $t \to_\beta^* v$ and $u \to_\beta^* v$. We refer to the *Archive of Formal Proofs* [3,5], for a recent, relatively simple, and machine checked proof of this fact.

[5] The point of the additional abstraction, is to turn the whole subterm into a value and thereby prevent its immediate reduction, which would cause a loop.

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] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. doi:10.2307/2371045.

[3] Bertram Felgenhauer, Julian Nagele, Vincent van Oostrom, and Christian Sternagel. The Z property. *Archive of Formal Proofs*, June 2016. https://www.isa-afp.org/entries/Rewriting_Z.shtml, Formal proof development.

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

[5] Julian Nagele, Vincent van Oostrom, and Christian Sternagel. A short mechanized proof of the Church-Rosser theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle. In *Proceedings of the 5th International Workshop on Confluence (IWC)*, 2016. arXiv:1609.03139.

[6] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996. http://www.cl.cam.ac.uk/~lp15/MLbook/pub-details.html.

[7] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.