The λ calculus

Frank Huch

Sommersemester 2018

Contents

1 Theoretical Foundations: the Lambda Calculus λ-calculus

The basic principle of functional programming languages is the λ-calculus, which was developed in 1941 by Church. His motivation was the development of a basic principle for mathematics and mathematical logic. In this context the notion of *computability* was formed. Its equivalence to Turing-machines was shown, which resulted in the **Church's Thesis** (or Church-Turing-Thesis).
Important aspects are:

- functions as objects
- bound and free variables
- rduction strategies

1.1 Syntax of the λ-calculus

Let Var be an enumerable set of variables, Exp the set of expressions in the pure λ-calculus, which are defined by ($e \in \text{Exp}$):

$$\begin{array}{rlll} e ::= & v & (\text{mit } v \in \text{Var}) & \text{variable} \\ | & (e\ e') & (\text{mit } e, e' \in \text{Exp}) & \text{application} \\ | & \lambda v.e & (\text{mit } v \in \text{Var}, e \in \text{Exp}) & \text{abstraction} \end{array}$$

We use the following conventions to omit brackets:

- Application binds left-associative, i.e. we write xyz instead of $((xy)z)$
- The scope of a λ reaches as far as possible. This means $\lambda x.xy$ means $\lambda x.(xy)$ and not $((\lambda x.x)y)$
- We use lists of variables in a abstraction instead of several nested λs: $\lambda xy.e$ instead of $\lambda x.\lambda y.e$

Note: There are no constants (predefined functions) or if-then-else structures. They all can be defined within the pure λ-calculus, as we will see later. As a consequence the pure λ-calculus is minimal and universal.

1.2 Substitution

The **semantics** of the pure λ-calculus is defined as known from functional programming languages: $\lambda x.e$ relatex to an anonymous function `\x -> e`. Hence, we get the semantics of a function application as $(\lambda x.x)z \rightsquigarrow z$, where in the body of the function (x) the variable x is substituted by z. Unfortunately, this rule cannot be realised that simple, as the following example shows. Substituting variables can result in name clashes:

$$\begin{array}{lcll} (\lambda f.\lambda x.fx)x & \not\rightsquigarrow & \lambda x.xx & \text{conflict} \\ (\lambda f.\lambda x.fx)x & \rightsquigarrow & \lambda y.xy & \text{no conflict} \end{array}$$

A prior free variable (which could be bound outside or can be seen as a constant) can be the application rule be substituted into a λ abstraction for the same varibale and hence, as a mistake get bound.
To formalise this, it is reasonable to first introduce the notions of *free* and *bound variables*:

$$\begin{array}{ll} \text{free}(v) = \{v\} & \text{bound}(v) = \emptyset \\ \text{free}((e\ e')) = \text{free}(e) \cup \text{free}(e') & \text{bound}((e\ e')) = \text{bound}(e) \cup \text{bound}(e') \\ \text{free}(\lambda v.e) = \text{free}(e) \setminus \{v\} & \text{bound}(\lambda v.e) = \text{bound}(e) \cup \{v\} \end{array}$$

An expression e is called *closed (or combinator)*, if $\text{free}(e) = \emptyset$.
Important: In an application we may only replace a variable by a parameter expression, if no free variable within the parameter expression gets bound by another λ abstraction. We precise this by the formal definition of *substitution*:
Let $e, f \in \text{Exp}$, $v \in \text{Var}$. Then the application of a *substitution* $e[v/f]$ (*replace* v *by* f *in* e) is defined by:

$$\begin{array}{rll} v[v/f] & = f & \\ x[v/f] & = x & \text{für } x \neq v \\ (e\ e')[v/f] & = (e[v/f]\ e'[v/f]) & \\ \lambda v.e[v/f] & = \lambda v.e & \\ \lambda x.e[v/f] & = \lambda x.(e[v/f]) & \text{for } x \neq v, x \notin \text{free}(f) \\ \lambda x.e[v/f] & = \lambda y.(e[x/y][v/f]) & \text{for } x \neq v, x \in \text{free}(f), \\ & & \quad y \notin \text{free}(e) \cup \text{free}(f) \end{array}$$

Particulary the last rule guarantees, that no free variable (of f) get bound by another λ. Instead we rename the variable bound in this λ to a fresh variable.
Note, that this definition is not really a function, since in the last rule an arbitrary fresh variable can be introduced to solve the name conflict. However, if the variable, which has to be substituted, does not occur in the expression, the expression stays unchanged: if $v \notin \text{free}(e)$, then $e[v/f] = e$.
Using substitutions, we can now formalise the reduction for function application.

1.3 Reduction Rules

We define *Beta-Reduction (β-Reduction)* as the following relation:

$$\rightarrow_\beta \subseteq \text{Exp} \times \text{Exp} \text{ with } (\lambda v.e)f \rightarrow_\beta e[v/f]$$

Example:

$$(\lambda f.\lambda x.fx)x \rightarrow_\beta \lambda y.xy$$

and

$$(\lambda f.\lambda x.fx)x \rightarrow_\beta \lambda z.xz$$

Hence, $\rightarrow_\beta$ is not *confluent*[1]*!* However, the names of the formal parameters are irrelevant for the meaning of a function. In other words, two syntactically different functions, e.g. $\lambda x.x$ and $\lambda y.y$ are semantically equivalent (both represent the identity function).
This semantical equivalence can be integrated in our reduction semantics by introducing reduction rules for the renaming of bound variables. The result is a confluent relation.
We define *Alpha-Reduction (α-Reduction)* as the following relation:

$$\rightarrow_\alpha \subseteq \text{Exp} \times \text{Exp} \text{ with } \lambda x.e \rightarrow_\alpha \lambda y.(e[x/y]) \text{ (if } y \notin \text{free}(e))$$

Examples:

$$\lambda x.x \rightarrow_\alpha \lambda y.y, \quad \lambda y.xy \rightarrow_\alpha \lambda z.xz, \quad \lambda y.xy \not\rightarrow_\alpha \lambda x.xx$$

We obtain $e \leftrightarrow_\alpha^* e'$ (*e and e' are α-equivalent*) if and only if e und e' can only be distinguished by the names of the occurring bound variables.
In the following we consider α-equivalent expressions to be equivalent, i.e. we calculate on α-equivalence-classes instead of λ-expressions.
So far, β-reduction can evaluate expressions. However, this is only possible in the outer-most position. Therefore, we extend β-reduction to arbitrary positions within the expressions (and similarly for α-Reduction):

$$\begin{array}{lll} \text{If } e \rightarrow_\beta e' & \text{then also} & ef \rightarrow_\beta e'f \\ & \text{and} & fe \rightarrow_\beta fe' \\ & \text{and} & \lambda x.e \rightarrow_\beta \lambda x.e' \end{array}$$

The last extension of β-reduction to the body of a λ-abstraction is often omitted, if one is only interested in pure computations. For confluence considerations and a theory for program equivalence it is, however, important.
Properties of β-reduction:

- The relation $\rightarrow_\beta$ is confluent.
- Every expression has at most one normalform, with respect to $\rightarrow_\beta$ (considering α-equivalence classes).

However, there are also expression without normal-forms. This is important for the equivalence with Turing-machine, and corresponds to non-termination. An example for an expression, which has no normalform is

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx)$$

It contains a so called *self-application*: $\lambda x.xx$. In the programming language Haskell this program would not be accepted, because of a type error.
For finding a normal-form, if it exists, strategies are used, as we already know from the evaluation of functional programs. We recapitulate these strategies in the context of the λ-calculus.

[1] A relation $\rightarrow^*$ is *confluent*, if for all u, v, w with $u \rightarrow^* v$ and $u \rightarrow^* w$ there exists z, such that $v \rightarrow^* z$ and $w \rightarrow^* z$.

A β-*redex* is a sub-expression of the form $(\lambda x.e)f$. Then a *reduction strategy* is a function from the set of expressions into the set of β-redexes. It expresses which redex is supposed to be evaluated in the next step. **Important reduction strategies** are:

- *Outermost-Strategy (LO, non-strict, normal-order-reduction)*:
  Choose the outermost redex
- *Innermost-Strategy (LI, strict, applicative-order-reduction)*:
  Choose the innermost redex

$$\begin{array}{ll}(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \to_{\mathrm{LI}} (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \\ (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \to_{\mathrm{LO}} z\end{array}$$

Since all function abstractions have arity one, redexes cannot be distinguished with respect to left/right. However, also in the context of the λ-calculus the strategies are called LI and LO.

Lemma: Let e' be a normal-form of e, i.e. $e \to_\beta^* e' \not\to_\beta e''$. Then there exists an LO-derivation from e to e'.

Hence, LO computes the normal-form, if it exists, in contrast to LI, which may not terminate, although a normal form exists.
An important aspect (for optimisations, transformations, verification) is the **equivalence of expressions**.[2] Intuitively, two expressions e and e' are equivalent, if e and e' can be exchanged in every context, without changing the semantics of the whole expression.
Examples:

- $\lambda x.x$ is equivalent to $\lambda y.y$
- $\lambda f.\lambda x.((\lambda y.y)f)((\lambda z.z)x)$ is equivalent to $\lambda f.\lambda x.fx$

It would be nice if we could show equivalence of two expressions by means of a kind of evaluation. α- and β-equivalence are unfortunately not expressive enough, as the following example shows (first in Haskellnotation): $(+1)$ is equivalent with $\lambda x.(+)\ 1\ x$. In every context, these two expressions will behave similarly. Applying both expressins to an argument z, we obtain:

$$(+1)\ z \triangleq (+)\ 1\ z \text{ and } (\lambda x.(+)\ 1\ x)z \to_\beta (+)\ 1\ z$$

But both expressions are not α- or β-equivalent. To express this equivalence, the theory of the λ-calculus is extended by *Eta-reduction (η-reduction)*:

$$\to_\eta \subseteq \mathrm{Exp} \times \mathrm{Exp} \text{ with } \lambda x.ex \to_\eta e \,(\text{if } x \notin \mathrm{free}(e))$$

There are the following two views to η-reduction:

[2] In the following we only consider the equivalence on terminating expressions (expressions, which do have a normal-form). Otherwise, it is not clear, what a result of a computation is and what exactly is supposed to be equivalent.

- η-reduction is an anticipated β-reduction:
  $(\lambda x.ex)f \to_\beta ef$, if $x \notin \text{free}(e)$.
- Extensionality: functions are equivalent, if they have a similar *function-graph* (their set of argument-result-pairs). If $fx \leftrightarrow^*_\beta gx$[3] holds, then $f \leftrightarrow^*_{\beta,\eta} g$ (with $x \in \text{free}(f) \cap \text{free}(g)$), since:

$$f \leftarrow_\eta \lambda x.fx \leftrightarrow^*_\beta \lambda x.gx \to_\eta g$$

There are some more relations, from which the *Delta-reduction (δ-reduction)* if important in the context of functional programming. It defines, how predefined functions can be integrated into the calculus. For instance, $(+)\ 1\ 2 \to_\delta 3$. Predefined functions and data types are, however, not necessary for the expressiveness of the pure λ-calculus. The can be defined within the λ-calculus, as we will sketch in the next chapter.
Summary of Reductions in λ-calculus:

- α-reduction: renaming of parameters
- β-reduction: function application
- η-reduction: elimination of redundant λ-abstractions
- δ-reduction: calculation with predefined function on predefined data types

1.4 data objects in the pure λ-calculus

Data types are objects with operations, idea here: represent these objects by closed λ-expressions and define fitting operations, in the sense of an abstract data type.
We start with the data type of **boolean values**: The objects are `True` and `False` and the most important operation is branching, the `if-then-else`-function:

$$\texttt{if\_then\_else}(b, e_1, e_2) = \begin{cases} e_1 & \text{, falls } \quad b = \texttt{True} \\ e_2 & \text{, falls } \quad b = \texttt{False} \end{cases}$$

The branching function is a projecting to the second or third argument. Hence, we can implement `True` and `False` as projection functions as well:

$$\begin{array}{rll} \texttt{True} \equiv & \lambda x.\lambda y.x & \text{take the first argument} \\ \texttt{False} \equiv & \lambda x.\lambda y.y & \text{take the second argument} \end{array}$$

Then the `if-then-else`-function can simply be realised as an application of the boolean value

$$\texttt{Cond} \equiv \lambda b.\lambda x.\lambda y.bxy$$

[3] To be more precise, $u \leftrightarrow^*_\beta v$ means that there exists a w, such that $u \to^*_\beta w$ and $v \to^*_\beta w$.

Example:

$$\begin{array}{lll} \texttt{Cond True}\ e_1e_2 & \equiv (\lambda bxy.bxy)(\lambda xy.x)e_1e_2 & \to_\beta^3 (\lambda xy.xe_1e_2) \to_\beta^2 e_1 \\ \texttt{Cond False}\ e_1e_2 & \equiv (\lambda bxy.bxy)(\lambda xy.y)e_1e_2 & \to_\beta^3 (\lambda xy.ye_1e_2) \to_\beta^2 e_2 \end{array}$$

The next task is a represenation of **natural numbers**: We use the encoding as *Church-Numerals*: a number $n \in \mathbb{N}$ is represented as a functional, that applies a given Function f exactly n times to another given argument:

$$\begin{aligned} 0 &\equiv \lambda f.\lambda x.x \\ 1 &\equiv \lambda f.\lambda x.fx \\ 2 &\equiv \lambda f.\lambda x.f(fx) \\ 3 &\equiv \lambda f.\lambda x.f(f(fx)) \\ n &\equiv \lambda f.\lambda x.f^n x \end{aligned}$$

For this representation it is possible to define functions for computation:

- The most important function is the successor function `succ`:

  $$\texttt{succ} \equiv \lambda n.\lambda f.\lambda x.nf(fx)$$

  Example: we compute the successor of one:

  $$\begin{aligned} \texttt{succ}\ 1 &\equiv (\lambda n.\lambda f.\lambda x.nf(fx))(\lambda f.\lambda x.fx) \\ &\to_\beta \lambda f.\lambda x.(\lambda f.\lambda x.fx)f(fx) \\ &\to_\beta \lambda f.\lambda x.(\lambda x.fx)(fx) \\ &\to_\beta \lambda f.\lambda x.f(fx) \equiv 2 \end{aligned}$$

- For branching, it is also important to have a test, whether a number is zero or not:

  $$\texttt{is\_Null} \equiv \lambda n.n(\lambda x.\texttt{False})\texttt{True}$$

  Example: :

  $$\begin{aligned} \texttt{is\_Null}\ 0 = (\lambda n.n(\lambda x.\texttt{False})\texttt{True})(\lambda f.\lambda x.x) \\ \to_\beta (\lambda f.\lambda x.x)(\lambda x.\texttt{False})\texttt{True} \\ \to_\beta (\lambda x.x)\texttt{True} \\ \to_\beta \texttt{True} \end{aligned}$$

1.5 Expressiveness of the λ-calculus

It remains to show, that the pure λ-calculus is computational universal. We do not prove this here, but argue for this by showing, that recursion can be realised, which is formalised with the folowing **Fixed-point theorem**:

Lemma: For every $F \in \text{Exp}$ there exists an expression X, such that $FX \leftrightarrow_\beta^* X$.

Proof: For instance, choose $X = YF$ with *Fixed-point combinator* Y:

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

It remains to proof, that $YF \leftrightarrow_\beta F(YF)$ holds. Exercise.
The pure λ-calculus is minimal, but computational universal. However, his relevance is more of theoretical nature (Barendregt 1984). We do not want to prove this here. The represenation of booleans, natural numbers and recursion, should however illustrate, a universal programming is possible in the pure λ-calculus.
Although self application is not possible in Haskell we can implement the Y-combinator be means of a record definition:

```
newtype Fix a = Fix { app :: Fix a -> a }

fix = \ f -> (\ x -> f (app x x)) (Fix (\ x -> f (app x x)))
```

To show, how the Y-combinator (from pure λ-calculus) can be used to realised a recursive function definiton, we implement the factorial function using this `fix` function. For simplicity, we use Haskell numbers instead of Church numerals and Haskells data type `Bool`.

```
fac = fix (\ f -> \ x -> if x==0 then 1 else x*f (x-1))
```

We define a functional for the factorial function using the Y-combinator, which least fixed-point can be computed at any possible argument position.
Within the pure λ-calculus, this program can be defined similarly. Unfortunately, the definition for the predecessor function is not straight forward (see exercises). Although, we were able to represent the Y-combinator in Haskell, there occur further type problems, implementing this example and similar examples with Church numerals in Haskell. Hence not every program from pure λ-calculus can be realised within Haskell.
For programming in the pure λ-calculus, we provide an interpreter in the iLearn page.

1.6 Enriched λ-calculus

For functional programming, an enriched λ-calculus, which also contains constants (predefined objects and functions), `let`, `if-then-else` and a built in fixed-point combinator, is more relevant.
Syntax of the enriched λ-calculus:

$e ::=$	v	Variable
$\mid$	k	constant symbols
$\mid$	$(e\ e')$	applikation
$\mid$	`let` $v = e$ `in` e'	local definitions
$\mid$	`if` e `then` e_1 `else` e_2	alternative (sometimes also `case`)
$\mid$	$\mu v.e$	fixed-point combinator

The operational **semantics** consists of α-, β- and η-reduction and the following rules

- δ-reduction for constants, z.B. $(+)\ 21\ 21 \to_\delta 42$
- $\texttt{let}\ v = e\ \texttt{in}\ e' \to e'[v/e]$
- $\texttt{if True then}\ e_1\ \texttt{else}\ e_2 \to e_1$
  $\texttt{if False then}\ e_1\ \texttt{else}\ e_2 \to e_2$
- $\mu v.e \to e[v/\mu v.e]$

Example: Factorial function:

$$\texttt{fac} \equiv \mu f.\lambda x.\ \texttt{if}\ ((==)\ x\ 0)\ \texttt{then}\ 1\ \texttt{else}\ ((*)\ x\ (f\ ((-)\ x\ 1)))$$

The enriched λ-calculus is the basis for the **implementation of functional languages** (Peyton, Jones 1987):

1. Programs are translated into this calculus (Core-Haskell in `ghc`):
   - Pattern-Matching is translated into `if-then-else` (or `case`),
   - $f\ x_1\ \ldots\ x_n = e$ is translated into $f = \lambda x_1 \ldots x_n.e$,
   - and recursive Functions are realised by a special fixed-point combinator.
   - A Program is then a list of `let`-declarations and an expression, which is supposed to be evaluated.
2. The calculus is then implemented by means of a special abstract Machine, e.g. the SECD-Machine by Landin in `ghc` or by means of other graph-reduction-machines.

Further Applications of the enriched λ-calculus are denotational semantics and type theory (typed λ-calculus).
The fixed-point combinator `fix`, is also available in Haskell (library `Data.Function`) and can be used to compute the fixed-point of a functional. Since Haskell provides recursion, it can be defined easier, than using self-application:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

It can be used to implement recursive functions, as we used `fix` in the definition of the factorial function above.