The λ-calculus

Hans

September 2023

The λ-calculus is the model of computation underlying Haskell and other functional programming languages. In this and the following short notes I will sketch the fundamentals of it.

The λ-calculus is a tiny language that has the language constructs that any functional programming language must have: The ability to define functions and the ability to call a function with an argument. Every functional programming language contains a version of the λ-calculus.

Syntax First let us consider the pure λ-calculus that has no pre-defined constants and no notion of type. We consider a version in which expressions have the abstract syntax given by the formation rules

$$e ::= \underbrace{x}_{\text{variables}} \mid \underbrace{\lambda x.e_1}_{\text{abstraction}} \mid \underbrace{e_1 e_2}_{\text{application}}$$

In this version of the language there are no numbers, truth values or anything else (we will do this later in the course) . We can still define interesting functions.

Example 1. The identity function can be written as $\lambda x.x$.

We can use parentheses whenever needed. The intention is that $\lambda x.e_1$ denotes a function with formal parameter x and body e_1. The scope of x is **all of** e_1. We can introduce parentheses to delimit the scope, if needed. The intention is that $e_1 e_2$ denotes that the function e_1 is to be applied to the argument e_2.

Scoping In an abstraction $\lambda x.e$, the x is called a *binding occurrence* of x. If the same x occurs inside e, we call it a *bound occurrence* of x. A binding occurrence should be thought of as a placeholder; its name is not important. For instance, $\lambda x.x$ and $\lambda y.y$ are both expressions the describe the identity function. In $\lambda x.x$, the first x is the binding occurrence (there can be at most one binding occurrence of the name in an abstraction) and the second is a bound occurrence (there can be many such bound occurrences). In order not to complicate the semantics and the notion of substitution, *we always assume that all binding occurrences use different variable name*s. If need be, we systematically rename the binding occurrences and the bound names. If an expression e_2 can be obtained from some other expression e_1 by systematically renaming zero or more binding occurrences and their bound occurrences, we say that e_1 and e_2 are *α-convertible* and write $e_1 \equiv_\alpha e_2$.

Semantics We can give the λ-calculus a small-step operational semantics in which transitions are of the form

$$e \to e'$$

read: Expression e reduces to expression e'. We now define this relation, which we call the *reduction relation.*

The *β-rule* describes how we evaluate an application $e_1 e_2$, when e_1 is an abstraction.

$$(\textsc{Beta}) \qquad (\lambda x.e_1)e_2 \Rightarrow e_1[x \mapsto e_2].$$

The notation $e_1[x \mapsto e_2]$ means that the actual parameter e_2 is substituted for each occurrence of the formal parameter in the body of e_1.

We call a subexpression for which a reduction is possible a *redex*; if the reduction is a β-reduction we speak of a β-redex.

The other reduction rules describe how an application will behave if we do not perform a beta-reduction and that α-convertible expressions have the same reductions.

$$(\textsc{Left}) \quad \frac{e_1 \Rightarrow e_1'}{e_1 e_2 \Rightarrow e_1' e_2} \qquad (\textsc{Right}) \quad \frac{e_2 \Rightarrow e_2'}{e_1 e_2 \Rightarrow e_1 e_2'}$$

$$(\textsc{Alpha}) \quad \frac{e_1 \equiv_\alpha e_2 \quad e_1' \equiv_\alpha e_2' \quad e_2 \Rightarrow e_2'}{e_1 \Rightarrow e_1'}$$

So we can perform reduction in any subterm of an application but *not* underneath an abstraction, since there are no rules for that.

Example 2. Here is an example reduction.

$$
\begin{aligned}
(\lambda x.\lambda y.yx)\underbrace{((\lambda z.zz)y)}_{\text{redex}} &\Rightarrow (\lambda x.\lambda y.yx)(yy) \\
&\equiv_\alpha \underbrace{(\lambda x.\lambda w.wx)(yy)}_{\text{redex}} \\
&\Rightarrow \lambda w.wyy
\end{aligned}
$$

If we did not have alpha-conversion, we would run into the problem of name clashes since there are two different y's in the original expression – one is bound, the other is free.