λ-calculus

1 Introduction

See also the video lecture recording: Introduction to lambda-calculus on Canvas.

Many programming languages are used in practice, e.g. Java, Python, C, Haskell etc. There are also theoretical programming languages, called *calculi*. They are usually very small and simple, which makes them painful for programming but valuable for analyzing and understanding specific aspects of computation.

For example, Turing machines are programs, so the definition of Turing machine and the associated execution rules constitute a programming language. It is an imperative language with just 6 instructions. Bad for programming, great for addressing fundamental questions of computability and complexity.

Another calculus, which you learnt at primary school, is *arithmetic*. This is a calculus of numbers. It has expressions, such as $(3 + 4) \times 2$, and reduction rules:

$$(3 + 4) \times 2 \rightsquigarrow 7 \times 2 \rightsquigarrow 14$$

In the 1930s Alonzo Church invented λ-calculus as a calculus of functions. It later became the basis for *functional languages* such as OCaml and Haskell. But it's also used for studying imperative programming, because they too use functions, parameter passing etc.

λ-calculus uses two notations.

- λx means "the function that maps x to".
- Juxtaposition means application.

Let's combine this with arithmetic. For example, $\lambda x.\, x{+}\underline{3}$ is the function that adds 3, so $(\lambda x.\, x{+}\underline{3})5$ represents the number 8.

I'm including the underline to distinguish an integer n from the corresponding term. (Often omitted in practice.) Expressions are also called *terms*. An expression of the form $\lambda x.M$ is called a λ-*abstraction*. We call x the *formal argument* and M the *body*.

Exercise What number does $(\lambda x.\, \underline{7} \times x + \underline{5})\, \underline{2}$ represent?

Exercise What number does $((\lambda f.\, \lambda x.\, f(f(x)))(\lambda x.\, x + \underline{3}))\, \underline{17}$ represent?

2 Syntax

See also the video lecture recording: Syntax of lambda-calculus on Canvas.

The BNF syntax of λ-calculus with arithmetic is as follows. We assume a countably infinite set Vars of variables.

$$\begin{array}{rcl} M & ::= & x \ \ (x \in \mathsf{Vars}) \\ & & \mid \lambda x.M \ \ (x \in \mathsf{Vars}) \\ & & \mid MM \mid \\ & & \underline{n} \ \ (n \in \mathbb{Z}) \\ & & \mid M + M \mid M - M \mid M \times M \end{array}$$

As with arithmetic or regular expressions, we need precedence conventions to be able to correctly parse a λ-term.

- Application has highest precedence.
- λx has lowest precedence.
- Arithmetic operations are intermediate.
- Application associates to the left.

The last point means that $M\,N\,P$ should be bracketed as $(M\,N)\,P$ not as $M\,(N\,P)$.

Exercise Show the implicit brackets in $(\lambda x.\,\lambda y.\,z\,x\,y)\,\underline{15}\,\underline{12}$.

In the above term, the occurrence of z is *free*, whereas the occurrence of x and y after the dots) are bound. Specifically, the occurrence of x is bound to the binder λx and the occurrence of y to the binder λy. The rule is that an occurrence of a variable is bound to the innermost binder whose scope contains the occurrence. If there is no such binder, the occurrence is free. For example, in $\lambda x.\,\lambda x.\,\lambda y.\,x + \underline{3}$, the occurrence of x after the dots is bound to the second λx.

It is a general principle that the choice of bound variables doesn't matter. So let's say that terms are α-*equivalent* when they are the same except for the binders and bound variables, and all bound variables are bound to the same place.

Exercise Consider the terms $\lambda x.\,\lambda x.\,x + \underline{3}$ and $\lambda u.\,\lambda v.\,\lambda u.\,v + \underline{3}$. Are they α-equivalent?

Exercise Consider the terms $\lambda x.\,\lambda x.\,\lambda y.\,x + z$ and $\lambda u.\,\lambda v.\,\lambda u.\,v + w$. Are they α-equivalent?

We regard α-equivalent terms as the same. The importance of this will soon be evident.
A term with no free variables is said to be *closed*.

3 Evaluation

See also the video lecture recording: Evaluation of lambda-terms on Canvas.

We evaluate a λ-term using δ-reductions and β-reductions.

- A δ-*reduction* is just an ordinary arithmetical reduction.

$$\underline{m} + \underline{n} \ \leadsto_\delta \ \underline{m+n} \tag{1}$$
$$\underline{m} - \underline{n} \ \leadsto_\delta \ \underline{m-n} \tag{2}$$
$$\underline{m} \times \underline{n} \ \leadsto_\delta \ \underline{mn} \tag{3}$$

For example:

$$\underline{5} + \underline{7} \quad \leadsto_\delta \quad \underline{12}$$

- A *β-reduction* says how to apply a λ-abstraction:

$$(\lambda x.\, M)\, N \quad \leadsto_\beta \quad M[N/x] \tag{4}$$

  Here $M[N/x]$ indicates a substituted term, viz. M with N (the *actual argument*) substituted for x (the *formal argument*). For example:

$$(\lambda x.\, x + \underline{3})\underline{7} \quad \leadsto_\beta \quad \underline{7} + \underline{3}$$

Usually we just write $\leadsto$ rather than $\leadsto_\delta$ and $\leadsto_\beta$.

We can also evaluate inside a term. A formal way of saying this is that $\leadsto$ is defined inductively by the reductions (1)–(4) together with the following rules:

- If $M \leadsto M'$ then $\lambda x.\, M \leadsto \lambda x.\, M'$.
- If $M \leadsto M'$ then $M + N \leadsto M' + N$.
- If $N \leadsto N'$ then $M + N \leadsto M + N'$.
- Likewise for subtraction and multiplication.
- If $M \leadsto M'$ then $MN \leadsto M'N$.
- If $N \leadsto N'$ then $MN \leadsto MN'$.

Any term of the form $\underline{m} + \underline{n}$ or $\underline{m} - \underline{n}$ or $\underline{m} \times \underline{n}$ is called a *δ-redex*. Any term of the form $(\lambda x.M)N$ is called a *β-redex*.

4 Reduction graph

Often a λ-term can be reduced in more than one way. For example $(\underline{3} + \underline{5}) \times (\underline{2} + \underline{1})$ can be δ-reduced to $\underline{8} + (\underline{2} + \underline{1})$ or to $(\underline{3} + \underline{5}) \times \underline{3}$. The term's *reduction graph* shows all the possible ways of reducing.

$$\begin{array}{ccc} (\underline{3} + \underline{5}) \times (\underline{2} + \underline{1}) & \rightsquigarrow & \underline{8} \times (\underline{2} + \underline{1}) \\ \downarrow & & \downarrow \\ (\underline{3} + \underline{5}) \times \underline{3} & \rightsquigarrow & \underline{8} \times \underline{3} \rightsquigarrow \underline{24} \end{array}$$

Exercise Write out the reduction graph of $(\lambda x.\, x + (\lambda y.\, y + \underline{1})\underline{3})\underline{2}$.

Exercise Write out the reduction graph of $(\lambda x. x + \underline{3})((\lambda x.\, x + \underline{7})\underline{1})$.

Typically a programming language has a fixed evaluation order. For example, OCaml would start the previous exercise by evaluating the argument, whereas Haskell would start by performing the outermost β-reduction.

5 Take care with substitution

There are two potential pitfalls when substituting.

1. When we reduce $(\lambda x.\,((\lambda x.\,x)17) + x)8$ using outermost β-reduction, it's important to only replace *free* occurrences of x by 8. It's safer to first α-convert the term to $\lambda x.((\lambda y.\,y)17 + x)8$.

2. When we reduce $(\lambda y.(\lambda x.x + y)7)(x + 5)$ using outermost β-reduction, we must not allow the free occurence of x in the argument to be captured. It's safer to first α-convert the term to $(\lambda y.(\lambda z.z + y)7)(x + 5)$

6 Normal Forms

See also the video lecture recording: The normal form of a lambda-term on Canvas.

We stop evaluating when we reach a term that doesn't contain a δ-redex or a β-redex. Such a term is called a *$\beta\delta$-normal form*. Examples are $\underline{23}$ and $x + \underline{23}$ and $\lambda x.\,x + \underline{23}$. If we reduce a closed term, we might obtain $\underline{23}$ or $\lambda x.\,x + \underline{23}$ but we can't obtain $x + \underline{23}$.

7 Uniqueness of normal form

Is it possible that the choice of evaluation order may lead to different normal forms? The answer is No. Let's write $\rightsquigarrow^*$ for the reflexive transitive closure of $\rightsquigarrow$, so $M \rightsquigarrow^* N$ means that M reduces to N in zero or more steps. The *Church-Rosser theorem* says that if $M \rightsquigarrow^* N$ and $M \rightsquigarrow^* P$, then there is Q such that $N \rightsquigarrow^* Q$ and $P \rightsquigarrow^* Q$. So if M reduces to normal forms N and P, they must be equal.

This means that, for programming languages based on λ-calculus, the evaluation order won't affect the final answer: it's not possible that Haskell will give you one answer and OCaml will give you another. However, it's possible that Haskell will give you an answer and OCaml will give you nothing.

To see this, look at the following term:

$$\Omega \stackrel{\text{def}}{=} (\lambda x.\,xx)(\lambda x.\,xx)$$

It β-reduces to itself, so it never reaches normal form. Now consider the term $(\lambda x.\,3)\Omega$. If the argument is evaluated first, it never terminates. But if the β-reduction is performed first, we get the answer 3.

8 Exercises

1. Draw a reduction graph for the following term:

$$(\lambda x.x + 3)\,((\lambda y.y \times 2)\,7)$$

2. Draw a reduction graph for the following term:

$$(\lambda x.\lambda y.\; x + 2 \times y)\,((\lambda x.x + 7)\,3)\,5$$

3. Draw a reduction graph for the following term:

$$(\lambda f.f2)(\lambda x.(x+3)+1)$$

4. Reduce the following term to normal form:

$$(\lambda y.y\,((\lambda x.y(yx))\,5))\,\lambda x.x \times 3$$