Introduction to the λ-calculus for CSci 4651

Elena Machkasova, UMN Morris

1 Overview

The Lambda Calculus (λ-calculus) was developed by an American mathematician Alonzo Church in 1930s to study computation. It uses anonymous functions and their composition to represent computation. The λ-calculus has been proven equivalent to the Turning Machine model of computation (invented by Alan Turing in 1936) which includes an infinite tape (modeling computer memory) and a collection of states of the computing device that perform different actions (modeling a program).

The λ-calculus and its more modern variants and extensions is used to formally define evaluation of a program, and specifically different orders of evaluation, including *call-by-value* and *call-by-name* evaluation strategies that we cover below, as well as *call-by-need* that represents lazy evaluation strategy, *π-calculus (pi-calculus)* that represents a distributed system of processes communicating via channels, and many other ones that you are welcome to look up if curious.

2 Syntax

λ-calculus operates with expressions known as λ-terms that we define below. Even though the classical λ-calculus doesn't directly include numbers, it is possible to define λ-terms that correspond to numbers and define arithmetic operations on them. For simplicity we include integer constants directly into expressions in the examples.

The syntax of the λ-calculus is given by a BNF-like notation. Expressions in the λ-calculus are called *terms*, and a term is recursively defined as follows:

$$M ::= x \mid c \mid \lambda x.M \mid M_1 M_2,$$

which means that a term M is either a variable (represented by x, but other lower-case letters can be used, such as y, z, u, etc.), a constant (a number or a boolean or another type, as needed), or $\lambda x.M$, where M is a term, or an application of one term to another, denoted $M_1 M_2$. Capital letters, in particular N, P, Q, are also used to denote terms.

$\lambda x.M$ represents an anonymous single-parameter function with the parameter x and the body M. For example, $\lambda x.x$ is the identity function that takes a parameter x and returns it unchanged. There is no way to represent a function with more than one parameters, but you can always "feed" the parameters separately, one by one: $\lambda x.\lambda y.M$ (an approach known as "currying").

$M_1 M_2$ stands for applying the term M_1 (as a function) to M_2. For example, $(\lambda x.x)5$ stands for applying the identity function to a constant 5. Note that the scope of the lambda extends as far to the right as possible, so parentheses are needed here since without the parentheses $\lambda x.x\ 5$ would stand for a function that takes x and returns the result of applying it to 5. Application is left-associative, so $M_1M_2M_3$ stands for $(M_1M_2)M_3$.

In addition to the operations given in the BNF above I will use arithmetic operations in examples. For instance, $\lambda x.x + 5$ represents a function that takes x and returns $x + 5$.

3 α-equivalence

Just like a function may use local and global variables, a lambda term may have *bound* and *free* variables. For example, in $\lambda x.x + y$ the variable x is bound by the lambda, and y is free.

Note that if we wrap this term into a lambda binding y, then both x and y will be bound, although by different lambdas: $\lambda y.\lambda x.x + y$.

Also observe that if we wrap $\lambda x.x+y$ into another lambda binding x, we get the term $\lambda x.\lambda x.x + y$ in which y is still free, and x is still bound by the inner lambda. The outer lambda is not using its argument. This means that its name can be anything (other than y), and the term will behave in exactly the same way, for example $\lambda z.\lambda x.x + y$.

Two terms M and N are called *α-equivalent* if they are equivalent to each other up to consistent renaming of the bound variables. For instance, $\lambda x.\lambda x.x+y$ is α-equivalent to $\lambda z.\lambda x.x + y$, but not to $\lambda y.\lambda x.x + y$. We say that adding λy to $\lambda x.x + y$ *captures* the free variable y.

α-equivalence is the same notion as a consistent renaming of local variables in a function or a method in a program or variables in expressions with $\forall$ and $\exists$ quantifiers in predicate calculus. As such, it is a familiar concept, and we omit the formal definition.

4 β-reduction

The λ-calculus describes computations in a functional language, so its main operation is a function application, i.e. applying one term to another. This operation is known as *β-reduction.* There are two main variations of the β-reduction: *call-by-name*, often abbreviated CBN, and *call-by-value* (CBV). Since this is the main operation that affects the entire system, we distinguish between the CBN λ-calculus and CBV λ-calculus.

In both cases of the calculi the goal of β-reduction is to *evaluate* a given term to a *value*, i.e. the final result, or the point where no more evaluation is possible. Strictly speaking, not all final results are values - for example, a computation may end up at an error, like $2\lambda x.x$ (here 2 is applied to $\lambda x.x$, but 2 is not a function, and therefore cannot be applied, so there is no way to evaluate this term any further). A more general term for the result of a computation that can't be evaluated any further is a *normal form*, and values are "good" normal forms. We define a value formally in section 4.2.

Note that in our examples in addition to β-reduction we are allowed to perform arithmetic operations, such as evaluate $2+3$ to 5. Such operations are allowed in both calculi and work the same way.

β-reduction is used for two purposes: as an *evaluation strategy* that defines how a program will be evaluated on the computer, and as an *optimization*, typically thought of as being performed at compilation time to speed up the program. The distinction between the two is that *evaluation strategy* is deterministic: given a term that needs to be evaluated, it will perform steps in a particular order to reach the result. β-reduction as an optimization, however, could be performed anywhere in a term at any point in a hope of simplifying the term and avoiding more work later. When we study β-reduction as an optimization, we are concerned with its *correctness*: is it possible that some operations that it performs change the result of evaluation? We discuss these issues more in section 4.3.

4.1 CBN λ-calculus

CBN β-reduction is recursively defined as follows:

$M_1 \overset{\beta_N}{\to} M_2$ if one of the following is true:

1. $M_1 = (\lambda x.M)N, M_2 = M[x := N]$, where $M[x := N]$ stands for M in which all free occurrences of x are replaced by N,

2. $M_1 = MN$, $M_2 = M'N$, and $M \overset{\beta_N}{\to} M'$,

3. $M_1 = MN$, $M_2 = MN'$, and $N \overset{\beta_N}{\to} N'$,

4. $M_1 = \lambda x.M, M_2 = \lambda x.M'$, and $M \overset{\beta_N}{\to} M'$.

The first case defines the β-reduction as a function application. The other three cases allow you to perform the reduction anywhere in a term.

Note that in order to get the correct result, $M[x := N]$ would often require parentheses around N as it's being substituted into M. See examples below for specifics.

Since we are allowing arithmetic operations (such as $+$, $\times$, etc.) as a part of the lambda calculus, we will additionally allow the β-reduction to recursively propagate into operands of an arithmetic operation:

Additionally, $M_1 \overset{\beta_N}{\to} M_2$ if

1. $M_1 = M\mathrm{op}N$, $M_2 = M'\mathrm{op}N$, and $M \overset{\beta_N}{\to} M'$ or

2. $M_1 = M\mathrm{op}N$, $M_2 = M\mathrm{op}N'$, and $N \overset{\beta_N}{\to} N'$,

where op is a binary arithmetic operation.

We will use $\to$ to mark a reduction step that is performing an arithmetic operation, and not a β-reduction.

Consider the following examples:

- $(\lambda x.x+x)2 \overset{\beta_N}{\rightarrow} 2+2 \rightarrow 4$. In this case we applied the function $\lambda x.x+x$ to 2 by replacing all occurrences of x in the body of the function $x+x$, and then performed the addition to get the value 4.

- $(\lambda x.\lambda y.y+x)2 \overset{\beta_N}{\rightarrow} \lambda y.y+2$. Here the result of the function application is a function $\lambda y.y+2$ (a function that adds 2 to its argument).

- $(\lambda x.x+x)((\lambda y.y+1)5) \overset{\beta_N}{\rightarrow} (\lambda x.x+x)(5+1)$ In this case the second term in the application gets reduced, corresponding to case 3 of the definition of $\overset{\beta_N}{\rightarrow}$ above. The result is not yet a normal form, you can to finish the evaluation on your own.

- There may be more than one possibility for a β-reduction step from the same term. For example, another possible step of β-reduction for the term above is $(\lambda x.x+x)((\lambda y.y+1)5) \overset{\beta_N}{\rightarrow} (\lambda y.y+1)5+(\lambda y.y+1)5$ (which is also not a normal form).

- $\lambda z.(\lambda y.y+2)1 \overset{\beta_N}{\rightarrow} \lambda z.1+2 \rightarrow \lambda z.3$. Here the β-reduction is applied inside a function. The final result cannot be reduced any further; it's a function that takes a single argument and returns 3.

- $(\lambda x.2\times x)5-(\lambda x.x+x)2 \overset{\beta_N}{\rightarrow} (2\times 5)-(\lambda x.x+x)2 \overset{\beta_N}{\rightarrow} (2\times 5)-(2+2) \rightarrow 10-4 \rightarrow 6$. This example illustrates using β-reduction together with arithmetic operations.

- Evaluation in the λ-calculus is not guaranteed to stop, it may go on infinitely (similar to an infinite loop in a program). Consider the term $(\lambda x.xx)(\lambda x.xx)$. In one step of β-reduction it evaluates to the same term: $(xx)[x := \lambda x.xx]$ is exactly $(\lambda x.xx)(\lambda x.xx)$. This term is often denoted as Ω. Terms for which evaluation never stops are known as *diverging terms.*

There are cases when $(\lambda x.M)N \overset{\beta_N}{\rightarrow} M[x := N]$ cannot be applied directly because of a *variable capture*, for example:

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

Here the substitution would work as following: $(\lambda y.xy)[x := y] = (\lambda y.yy)$. Observe that y was free in the given expression and should remain free in the result (presumably since it's a global variable that has some value or meaning in the global context). However, after the substitution y is bound by λy and became the same as the function parameter y.

This issue can be resolved using α-equivalence. Since the names of local (bound) variables don't matter, we will use a term that's α-equivalent to the given one, but doesn't cause variable capture: $(\lambda x.\lambda z.xz)y$. The term has the exact same functionality as the original one, but now the substitution is safe:

$$(\lambda x.\lambda z.xz)y \overset{\beta_N}{\rightarrow} \lambda z.yz$$

Here y remains free (global). Replacing a term by an α-equivalent one (usually to avoid a free variable capture) is known as *α-renaming*.

For all our examples we assume that α-renaming should be used to prevent a variable capture. One possible approach is to always rename all terms in a way that makes all bound names unique and distinct from all free variables names (this is known as the *distinct variables convention*). However, it becomes cumbersome to always rename bound variables in all examples, so we will just require α-renaming to prevent variable capture.

4.2 CBV λ-calculus

Consider the following CBN β-reduction:

$$(\lambda x.x + x)((\lambda y.y + 1)5) \overset{\beta_N}{\to} (\lambda y.y + 1)5 + (\lambda y.y + 1)5$$

Here the function $(\lambda x.x + x)$ is applied to the argument $(\lambda y.y + 1)5$, and the substitution takes the entire argument and substitutes it into the body of the function as is. However, the argument itself is not fully evaluated. It can be reduced as following: $(\lambda y.y + 1)5 \overset{\beta_N}{\to} 5 + 1 \to 6$.

If you were applying a function to an argument in a program, the argument would've been fully evaluated first, and then the resulting value would've been passed to the function. Only macros take their arguments as is, unevaluated.

CBN λ-calculus is more suited to representing macros. CBV λ-calculus was specifically designed to represent functions and function calls. It restricts function calls (the base case of the β-reduction) only to cases when the argument is already a value.

In order to define CBV β-reduction, denoted $\overset{\beta_V}{\to}$, we first define a *value*, i.e. a "good" final result of evaluation:

$$V ::= x \mid c \mid \lambda x.M$$

This means that a value is either a single variable or a constant or a λ-abstraction, i.e. a function $\lambda x.M$. Note that this means that a function is a value, which is consistent with the key idea of a functional language: functions are first-class citizens, in particular they can be passed to functions and return from functions. The inclusion of variables as values is more for technical reasons.

Based on the definition of a value, we now define CBV β-reduction. Note that only the base case (case 1) changes; all the other cases just recursively allow the reduction to be performed inside terms. We include the cases for arithmetic operations here as well:

$M_1 \overset{\beta_V}{\to} M_2$ if one of the following is true:

1. $M_1 = (\lambda x.M)V, M_2 = M[x := V]$, where $M[x := V]$ stands for M in which all free occurrences of x are replaced by V. Note that here the argument of the application is restricted to be a value, as defined above.

2. $M_1 = MN$, $M_2 = M'N$, and $M \overset{\beta_V}{\to} M'$,

3. $M_1 = MN$, $M_2 = MN'$, and $N \overset{\beta_V}{\to} N'$,

4. $M_1 = \lambda x.M, M_2 = \lambda x.M'$, and $M \overset{\beta_V}{\to} M'$,

5. $M_1 = M\text{op}N$, $M_2 = M'\text{op}N$, and $M \overset{\beta_V}{\to} M'$,

6. $M_1 = M\text{op}N$, $M_2 = M\text{op}N'$, and $N \overset{\beta_V}{\to} N'$,

Just as for CBN, we assume that α-renaming is done to avoid variable capture.

Many, but not all, CBN examples above still work the same in CBV. Let us revisit the examples from above:

- $(\lambda x.x + x)2 \overset{\beta_V}{\to} 2 + 2 \to 4$. This example works the same in CBV since 2 is a value (a constant).
- $(\lambda x.\lambda y.y + x)2 \overset{\beta_V}{\to} \lambda y.y + 2$. This example also works the same, for the same reason.
- $(\lambda x.x + x)((\lambda y.y + 1)5) \overset{\beta_V}{\to} (\lambda x.x + x)(5 + 1)$. This works in CBV since we again are applying a function to a constant (a value). The recursive definition allows us to reduce the argument of an application (by case 3, just like in CBN).
- The next example $(\lambda x.x+x)((\lambda y.y+1)5) \overset{\beta_N}{\to} (\lambda y.y+1)5+(\lambda y.y+1)5$ **does not** work in CBV since it's passing an unevaluted argument $(\lambda y.y + 1)5$ to the function.
- $\lambda z.(\lambda y.y+2)1 \overset{\beta_V}{\to} \lambda z.1+2 \to \lambda z.3$ works in CBV since, again, the argument of a function is a constant, and case 4 of the definition allows us to evaluate inside a function.
- $(\lambda x.2 \times x)5 - (\lambda x.x + x)2 \overset{\beta_V}{\to} (2 \times 5) - (\lambda x.x + x)2 \overset{\beta_V}{\to} (2 \times 5) - (2 + 2) \to 10 - 4 \to 6$ also works the same in CBV. Here we are using cases 5 and 6 of the recursive definition of $\overset{\beta_V}{\to}$.
- The final example of a diverging term Ω also works in CBV and results in divergence: since $\lambda x.xx$ is a function, and functions are values, we can perform the same reduction in CBV as follows: $(\lambda x.xx)(\lambda x.xx) \overset{\beta_V}{\to} (\lambda x.xx)(\lambda x.xx) \overset{\beta_V}{\to} (\lambda x.xx)(\lambda x.xx)\ldots$

As another example of a reduction that works in CBN, but not in CBV, consider $(\lambda x.x + x)(5 - 2)$. In CBN it can be evaluated as follows:

$$(\lambda x.x + x)(5 - 2) \overset{\beta_N}{\to} (5 - 2) + (5 - 2) \to 3 + 3 \to 6.$$

In CBV you have to evaluate the argument to a value before passing it in to a function:

$$(\lambda x.x + x)(5 - 2) \overset{\beta_V}{\to} (\lambda x.x + x)3 \to 3 + 3 \to 6.$$

The results are, of course, the same, but the steps performed are different. See section 4.4 for more explanations.

4.3 Evaluation vs optimization

Recall that in the beginning of section 4 we mentioned that β-reduction can be used as a program evaluation or as a compiler optimization. The difference between the two is that program evaluation is *deterministic*, i.e. given a term evaluation will always perform the same sequence of steps to get it to a normal form, if possible (or go into the same divergence patterns if the term diverges). An optimization may be performed anywhere in a term in a hope of simplifying it and eliminating work done by evaluation at “run time”.

Evaluation is also known as *operational semantics* of a calculus since “semantics” means “meaning”, and the meaning of a term is given by what it eventually evaluates to, so operational semantics is a set of rules to find the meaning of terms.

For both CBN and CBV λ-calculi evaluation is defined as follows: if there is more than one reduction step available (either β-reduction or an arithmetic operation), perform the *left-most outer-most* one. It is also often the rule that if the only remaining reductions are *under a lambda*, they are *not performed* - we will follow this rule.

TBD

4.4 Confluence

TBD