# Lambda-calculus and programming language semantics

Thibaut Balabonski @ UPSay
Winter 2023
https://www.lri.fr/~blsk/LambdaCalculus/

# Chapter 6: implementing the $\lambda$-calculus

## 1 Interpretation

Following the operational semantics of the $\lambda$-calculus, we can define an *interpreter*, that is a function that takes as input a $\lambda$-term and returns the value computed by this term.

Terms:

$$\begin{array}{rcll} t & ::= & N & \text{integer constant} \\ & | & x & \text{variable} \\ & | & \lambda x.t & \text{abstraction} \\ & | & t\ u & \text{application} \end{array}$$

Values:

$$\begin{array}{rcl} v & ::= & N \\ & | & \lambda x.t \end{array}$$

**Call by value, small-step semantics.** We define a reduction relation $t \longrightarrow t'$ describing one step of reduction. Call-by-value evaluation: evaluate the argument before resolving a function application. The first rule only applies if the argument is a value $v$.

$$\frac{}{(\lambda x.t)\ v \longrightarrow t\{x \leftarrow v\}} \qquad \frac{t \longrightarrow_\beta t'}{t\ u \longrightarrow t'\ u} \qquad \frac{u \longrightarrow_\beta u'}{v\ u \longrightarrow v\ u'}$$

The last rule only applies if the left member of the application is a value $v$. This forces evaluation from left to right. Note that we never reduce under $\lambda$-abstractions, and that we do not expect to reach a free variable.

Evaluation is done through a sequence of such steps:

$$t \longrightarrow^* v$$

We could define an interpreter performing sequences of steps, following this definition. The cost of one step would contain:

- the cost of finding the next redex $(\lambda x.t)\ u$ (proportional to the depth of the redex),
- the cost of performing the substitution (proportional to the size of $t$) and managing required renamings,
- the cost of reconstructing the term after transforming the redex (possibly proportional to the depth of the redex).

This cost would then be repeated for each step. This is *very* inefficient.

**Call-by-value, big-step semantics.** We define an evaluation relation $t \Downarrow v$ defining the value $v$ resulting from the evaluation of $t$. The relation is reflexive on terms that are already values.

$$\frac{}{N \Downarrow N} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.s \qquad u \Downarrow v \qquad s\{x \leftarrow v\} \Downarrow w}{t\ u \Downarrow w}$$

As before, we never reduce under $\lambda$-abstractions, and that we do not expect to reach a free variable. Evaluation of an application requires evaluating the function and the argument, and then evaluating the function body, with access to the value of the argument.

Following these rules, we define an interpretation function eval.

$$\begin{array}{rcll} \mathsf{eval}(N) & = & N & \\ \mathsf{eval}(\lambda x.t) & = & \lambda x.t & \\ \mathsf{eval}(t\ u) & = & \mathsf{eval}(s\{x \longleftarrow \mathsf{eval}(u)\}) & \text{if } \mathsf{eval}(t) = \lambda x.s \end{array}$$

The costs of finding the next redex and of rebuilding the term are amortized. All the costs related to substitution are still there.

**Call-by-value, big-step semantics with environments.** Rather than actually performing substitutions, we consider a term (with free variables) together with an environment (which gives values to the free variables of the term). The evaluation relation now takes the form

$$e \vdash t \Downarrow v$$

and means "the term $t$ taken in the environment $e$ evaluates to $v$".

To correctly implement the scoping rules of variables in the $\lambda$-calculus, every value possibly containing free variables should be given together with the environment in which it has been defined. Thus, the "value" associated to a $\lambda$-abstraction $\lambda x.t$ is a *closure*, that is a pair $\langle \lambda x.t, e \rangle$ of the $\lambda$-abstraction and an environment $e$ providing values for its free variables.

The values and environments are now defined by:

$$\begin{array}{rcl} v & ::= & N \\ & | & \langle \lambda x.t, e \rangle \\ e & ::= & \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\} \end{array}$$

Evaluation rules are as follows. The value of a variables is looked up in the environment. The value of a $\lambda$-abstraction is a closure built with the current environment. When evaluating the body of a function, we use the environment provided by the closure, to ensure that all free variables are associated to the proper (lexically scoped) values. We write $\{e, x \mapsto v\}$ the extension of the environment $e$ with the mapping of the variable $x$ to the value $v$.

$$\frac{}{e \vdash N \Downarrow N} \qquad \frac{e(x) = v}{e \vdash x \Downarrow v} \qquad \frac{}{e \vdash \lambda x.t \Downarrow \langle \lambda x.t, e \rangle}$$

$$\frac{e \vdash t \Downarrow \langle \lambda x.s, e' \rangle \qquad e \vdash u \Downarrow v \qquad \{e', x \mapsto v\} \vdash s \Downarrow w}{e \vdash t\ u \Downarrow w}$$

Following these rules, we define a new interpretation function, that takes as inputs a $\lambda$-term and an environment.

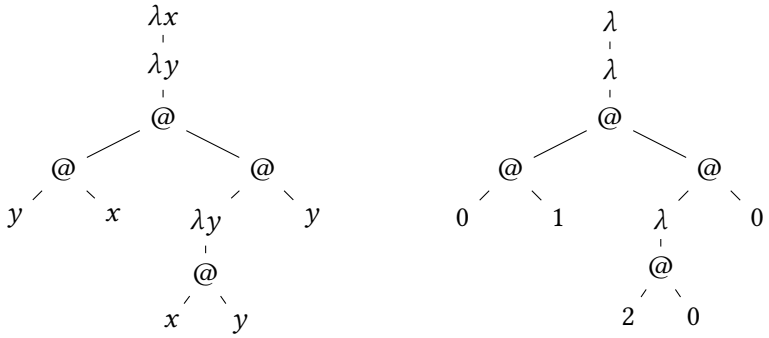
$$\begin{array}{rcll} \mathsf{eval}(N, e) & = & N & \\ \mathsf{eval}(x, e) & = & e(x) & \\ \mathsf{eval}(\lambda x.t, e) & = & \langle \lambda x.t, e \rangle & \\ \mathsf{eval}(t\ u, e) & = & \mathsf{eval}(s, \{e', x \mapsto \mathsf{eval}(u, e)\}) & \text{if } \mathsf{eval}(t) = \langle \lambda x.s, e' \rangle \end{array}$$

There is no substitution anymore. However we still have to deal with names.

**Call-by-value, big-step semantics with environments, without names.** We can represent $\lambda$-terms without using variables names, by replacing each variable occurrence by a *number* called *de Bruijn*

*index*, defined as the number of $\lambda$ that are between this variable occurrence and its binder.

$\lambda x.\lambda y.(y\ x\ ((\lambda y.xy)\ y))$ $\qquad$ $\lambda.\lambda.0\ 1\ ((\lambda.20)\ 0)$




With this nameless notations for terms, a $\beta$-redex now has the shape

$$(\lambda.t)\ u$$

and its reduction consists in replacing by $u$ the variable occurrences of $t$ that are bound by this $\lambda$. They are precisely: the occurrences of 0 that are not nested under any other $\lambda$, the occurrences of 1 that are nested under exactly 1 other $\lambda$, the occurrences of 2 that are nested under exactly 2 other $\lambda$'s... However, in our implementation we do not perform any substitution: the argument $u$ is just inserted at index 0 in the environment. More generally, an environment is now just a list of values, and each value is accessed through its index in the list.

Terms, values and environemnts. We write $\underline{n}$ for the variable with index $n$ (not to be mixed with the integer constant $N$). An environment is now written as a sequence of values (there are no names anymore, we refer to values in the environment only by their position).

| | | | |
|---|---|---|---|
| $t$ | $::=$ | $N$ | integer constant |
| | $\mid$ | $\underline{n}$ | variable (de Bruijn index) |
| | $\mid$ | $\lambda.t$ | abstraction |
| | $\mid$ | $t\ u$ | application |
| $v$ | $::=$ | $N$ | |
| | $\mid$ | $\langle \lambda.t, e \rangle$ | |
| $e$ | $::=$ | $v_0 \cdot v_1 \cdot \ldots \cdot v_k$ | |

Evaluation rules. We write $e[n]$ for taking the value of index $n$ in the list $e$. We write $v \cdot e$ for the environment obtained from by inserting $v$ at index 0 (and incrementing the indices of all the other values in $e$).

$$\frac{}{e \vdash N \Downarrow N} \qquad \frac{e[n] = v}{e \vdash \underline{n} \Downarrow v} \qquad \frac{}{e \vdash \lambda.t \Downarrow \langle \lambda.t, e \rangle} \qquad \frac{e \vdash t \Downarrow \langle \lambda.s, e' \rangle \qquad e \vdash u \Downarrow v \qquad v \cdot e' \vdash s \Downarrow w}{e \vdash t\ u \Downarrow w}$$

These rules give us a final version of our evaluation function, working on $\lambda$-terms in nameless representation.

$$\begin{aligned}
\mathrm{eval}(N, e) &= N \\
\mathrm{eval}(\underline{n}, e) &= e[n] \\
\mathrm{eval}(\lambda.t, e) &= \langle \lambda.t, e \rangle \\
\mathrm{eval}(t\ u, e) &= \mathrm{eval}(s, \mathrm{eval}(u, e) \cdot e') \qquad \text{if } \mathrm{eval}(t) = \langle \lambda.s, e' \rangle
\end{aligned}$$

**Efficient interpreter.** The final evaluation function can easily be translated into code, giving us an efficient interpreter (an interpreter from which the obvious inefficiencies have been removed). In the following caml code we assume a datatype `'a env` for environments containing elements of type `'a`, with functions `env_lookup: int -> 'a env -> 'a` for getting the element at a given index, and `env_add: 'a -> 'a env -> 'a env` for inserting an element at index 0. Many concrete data structures exist, which provide efficient implementations of these operations.

```
type term =
  | Cst of int
  | Var of int
  | Abs of term
  | App of term * term
type value =
  | Vint  of int
  | Vclos of term * value env

let rec eval e env = match e with
  | Cst n -> Vint n
  | Var n -> env_lookup n env
  | Abs t -> Vclos(t, env)
  | App(t, u) ->
     let Clos(s, env') = eval t env in
     let v = eval u env in
     eval s (env_add v env')
```

## 2 Compilation to abstract machine code

Instead of interpreting a program, we can translate it into a sequence of instructions for some machine. Two possibilities:

- we can target an actual microprocessor, and the instructions will be executed directly by the hardware,
- or we can target an *abstract machine*, which looks like a machine but does not correspond to actual hardware.

In general, the instructions of an abstract machine are chosen with two criteria: they are close to the basic mechanisms of the source language, and they can be implemented efficiently.

**Warm-up: an abstract machine for evaluating arithmetic expressions.** Arithmetic expression can famously be evaluated using a very simple machine working with a *stack* (a sequential data-structure in which we can add and remove elements "at the top").

Arithmetic expressions:

$$\begin{array}{rcl} a & ::= & N \\ & | & a_1 + a_2 \\ & | & a_1 \times a_2 \end{array}$$

Instructions:

- CONST($N$): push integer $N$ on top of the stack,
- ADD, MUL: pop two integers from the stack, combine, push the result back on the stack,
- $c_1$; $c_2$: execute $c_1$, then $c_2$.

An expression $a$ is translated into a sequence of instructions $\mathcal{C}(a)$ that evaluates $a$ and pushes the obtained value on the stack. Evaluating a binary operation consists in evaluating both parts, saving the intermediaite values on the stack, and then combining the two stored values.

$$\begin{array}{rcl} \mathcal{C}(N) & = & \text{CONST}(N) \\ \mathcal{C}(a_1 + a_2) & = & \mathcal{C}(a_1); \mathcal{C}(a_2); \text{ADD} \\ \mathcal{C}(a_1 \times a_2) & = & \mathcal{C}(a_1); \mathcal{C}(a_2); \text{MUL} \end{array}$$

The state of the machine executing such instructions is defined by:

- a code pointer, to the instructions that have to be executed,
- a stack with the stored intermediate results.

The execution of the machine is a sequence of steps, defined by the following table.

| State before | | State after | |
|---|---|---|---|
| code | stack | code | stack |
| CONST($N$); $c$ | $s$ | $c$ | $N \cdot s$ |
| ADD; $c$ | $N_1 \cdot N_2 \cdot s$ | $c$ | $(N_1 + N_2) \cdot s$ |

This machine can be implemented efficiently, either by interpreting its instructions, or by expanding them into native code for our actual processor.

**An abstract machine for call-by-value $\lambda$-calculus.** The "SECD" machine for executing $\lambda$-terms has three elements:

- a code pointer $c$, to the instructions that have to be executed,
- an environment $e$, which gives values to the variables,
- a stack $s$, which stores intermediate results and other useful context information.

In addition to the basic arithmetic operation, it features the following instructions:

- ACCESS($n$): push the $n$-th element of the environment on the stack,
- CLOSURE($c$): make a closure with code $c$ and the current environment, and push it on the stack,
- APPLY: pop an argument and a function closure, and perform the application,
- RETURN: returns to caller.

Compilation:

$$\begin{aligned} \mathcal{C}(\underline{n}) &= \text{ACCESS}(n) \\ \mathcal{C}(\lambda.t) &= \text{CLOSURE}(\mathcal{C}(t); \text{RETURN}) \\ \mathcal{C}(t\ u) &= \mathcal{C}(t); \mathcal{C}(u); \text{APPLY} \end{aligned}$$

Execution steps. The APPLY instruction records on the stack the context information that should be restored after the call. This information consists in the code pointer $c$ to the instructions that follow the code, and the current environment. Then the RETURN instruction retrieves and restores this information. We write again $\langle c, e \rangle$ for a closure with code $c$ and environment $e$ (note that here, $c$ is a sequence of instructions and not a $\lambda$-term).

| State before | | | State after | | |
|---|---|---|---|---|---|
| code | env | stack | code | env | stack |
| ACCESS($n$); $c$ | $e$ | $s$ | $c$ | $e$ | $e[n] \cdot s$ |
| CLOSURE($c'$); $c$ | $e$ | $s$ | $c$ | $e$ | $\langle c', e \rangle \cdot s$ |
| APPLY; $c$ | $e$ | $v \cdot \langle c', e' \rangle \cdot s$ | $c'$ | $v \cdot e'$ | $c \cdot e \cdot s$ |
| RETURN; $c$ | $e$ | $v \cdot c' \cdot e' \cdot s$ | $c'$ | $e'$ | $v \cdot s$ |

**Compilation, extended for (nameless) local variables.** Since it is used a lot, the definition of local variables using let is given a pair of dedicated instructions. The scope of the local variable is defined by a pair LET/ENDLET that adds the value to the environment, and removes it at the end.

$$\mathcal{C}(\text{let } t \text{ in } u) = \mathcal{C}(t); \text{LET}; \mathcal{C}(u); \text{ENDLET}$$

- LET: add to the environment the value taken at the top of the stack,
- ENDLET: drop a value from the environment.

Execution steps.

| State before | | | State after | | |
|---|---|---|---|---|---|
| code | env | stack | code | env | stack |
| LET; $c$ | $e$ | $v \cdot s$ | $c$ | $v \cdot e$ | $s$ |
| ENDLET; $c$ | $v \cdot e$ | $s$ | $c$ | $e$ | $s$ |

**Proving partial correctness of the machine.** Goal: if a term $t$ evaluates to $v$ in the $\lambda$-calculus (meaning: if $t \longrightarrow^* v$), then the machine started in the state $(\!| \mathcal{C}(t) \mid \varepsilon \mid \varepsilon |\!)$ terminates in a state $(\!| \varepsilon \mid \varepsilon \mid v' |\!)$ where $v'$ represents the value $v$.

We prove this by induction on the big-step semantics $e \vdash t \Downarrow v$. For the induction to work, we need to generalize our goal property in such a way that it will be applicable to the evaluation of subterms. For this, we include in the statement an arbitrary sequence of code, an environment and a stack. The property we prove is then:

$$\text{If } e \vdash t \Downarrow v \text{ then for any } c \text{ and } s \text{ we have } (\!| \mathcal{C}(t); c \mid \mathcal{T}(e) \mid s |\!) \longrightarrow^* (\!| c \mid \mathcal{T}(e) \mid \mathcal{T}(v) \cdot s |\!)$$

This property uses a translation function $\mathcal{T}(.)$ for translating values and environments from the $\lambda$-calculus in machine format.

$$\begin{array}{rcl} \mathcal{T}(N) & = & N \\ \mathcal{T}(\langle \lambda.t, e \rangle) & = & \langle \mathcal{C}(t); \mathsf{RETURN}, \mathcal{T}(e) \rangle \\ \mathcal{T}(v_0 \cdot v_1 \cdot \ldots \cdot v_k) & = & \mathcal{T}(v_0) \cdot \mathcal{T}(v_1) \cdot \ldots \cdot \mathcal{T}(v_k) \end{array}$$

Proof of the property, by induction over the derivation of $e \vdash t \Downarrow v$.

- Case $e \vdash N \Downarrow N$. Then $\mathcal{C}(N) = \mathsf{CONST}(N)$ and $\mathcal{T}(N) = N$ and we have

  $$(\!| \mathsf{CONST}(N); c \mid \mathcal{T}(e) \mid s |\!) \longrightarrow (\!| c \mid \mathcal{T}(e) \mid N \cdot s |\!)$$

  in one step of the machine.

- Case $e \vdash \underline{n} \Downarrow v$, with $e[n] = v$. Then $\mathcal{C}(\underline{n}) = \mathsf{ACCESS}(n)$ and $\mathcal{T}(e)[n] = \mathcal{T}(v)$ and we have

  $$(\!| \mathsf{ACCESS}(N); c \mid \mathcal{T}(e) \mid s |\!) \longrightarrow (\!| c \mid \mathcal{T}(e) \mid \mathcal{T}(v) \cdot s |\!)$$

  in one step of the machine.

- Case $e \vdash \lambda.t \Downarrow \langle \lambda.t, e \rangle$. Then $\mathcal{C}(\lambda.t) = \mathsf{CLOSURE}(\mathcal{C}(t); \mathsf{RETURN})$ and $\mathcal{T}(\langle \lambda.t, e \rangle) = \langle \mathcal{C}(t); \mathsf{RETURN}, \mathcal{T}(e) \rangle$ and we have

  $$(\!| \mathsf{CLOSURE}(\mathcal{C}(t); \mathsf{RETURN}); c \mid \mathcal{T}(e) \mid s |\!) \longrightarrow (\!| c \mid \mathcal{T}(e) \mid \langle \mathcal{C}(t); \mathsf{RETURN}, \mathcal{T}(e) \rangle \cdot s |\!)$$

  in one step of the machine.

- Case $e \vdash t\ u \Downarrow w$ with $e \vdash t \Downarrow \langle \lambda.r, e' \rangle$, $e \vdash u \Downarrow v$ and $v \cdot e' \vdash r \Downarrow w$. Then $\mathcal{C}(t\ u) = \mathcal{C}(t); \mathcal{C}(u); \mathsf{APPLY}$ and $\mathcal{T}(\langle \lambda.r, e' \rangle) = \langle \mathcal{C}(r); \mathsf{RETURN}, \mathcal{T}(e') \rangle$. We build the following sequence of execution of the machine:

| | | | | |
|---|---|---|---|---|
| | $\mathcal{C}(t); \mathcal{C}(u); \mathsf{APPLY}; c$ | $\mathcal{T}(e)$ | $s$ | |
| $\longrightarrow^*$ | $\mathcal{C}(u); \mathsf{APPLY}; c$ | $\mathcal{T}(e)$ | $\mathcal{T}(\langle \lambda.r, e' \rangle) \cdot s$ | by IH on $e \vdash t \Downarrow \langle \lambda.r, e' \rangle$ |
| $\longrightarrow^*$ | $\mathsf{APPLY}; c$ | $\mathcal{T}(e)$ | $\mathcal{T}(v) \cdot \mathcal{T}(\langle \lambda.r, e' \rangle) \cdot s$ | by IH on $e \vdash u \Downarrow v$ |
| $\longrightarrow$ | $\mathcal{C}(r); \mathsf{RETURN}$ | $\mathcal{T}(v) \cdot \mathcal{T}(e')$ | $c \cdot \mathcal{T}(e) \cdot s$ | |
| $\longrightarrow^*$ | $\mathsf{RETURN}$ | $\mathcal{T}(v) \cdot \mathcal{T}(e')$ | $\mathcal{T}(w) \cdot c \cdot \mathcal{T}(e) \cdot s$ | by IH on $v \cdot e' \vdash r \Downarrow w$ |
| $\longrightarrow$ | $c$ | $\mathcal{T}(e)$ | $\mathcal{T}(w) \cdot s$ | |

Remark: this correctness property is called *partial*, because it applies only to terms $t$ such that call-by-value evaluation $e \vdash t \Downarrow v$ succeeds. We could state a more precise property ensuring that other possible observable behaviours (non-termination, termination on error...) are also preserved, but the proof is significantly harder.

## 3 An abstract machine for call-by-name $\lambda$-calculus.

Call-by-name and call-by-need evaluation strategies do not use the same mechanisms than call-by-value. Machines for these strategies are thus also different.

**Call-by-name, small-step semantics.** The relation $t \longrightarrow t'$ describes individual reduction steps. Call-by-name evaluation: apply a function to its argument without evaluating the argument first, then evaluate the argument on demand.

$$\frac{}{(\lambda x.t)\ u \longrightarrow t\{x \leftarrow u\}} \qquad \frac{t \longrightarrow t'}{t\ u \longrightarrow t'\ u}$$

Differences with call-by-value reduction rules: the argument of a $\beta$-redex is not constrained anymore to be a value. Right members of applications are not evaluated. As for call-by-value, we will actually work with an environment rather than substitutions, and use the nameless representation based on de Bruijn indices.

The KAM (*Krivine Abstract Machine*) uses the same three kinds of elements than the SECD machine: a code pointer, an environment and a stack. Their contents is slightly different however.

- The environment still contains the actual parameters provided to a function, but these parameters are not evaluated anymore. Then the environment contains "terms to evaluate later" rather than values. These elements, called *thunks*, are pairs made of a code pointer and an environment (a thunk is similar to a functional closure, but its "code" part is not necessarily the code of a function).
- The stack contains parts of the term that are waiting to be considered, in particular the arguments to which a function has not been applied yet. Once again, these elements are thunks (and this contrasts with the stack of the SECD, which contained intermediate values).

The KAM has only three instructions. Compilation is as follows.

$$\begin{array}{rcl} \mathcal{C}(\underline{n}) & = & \mathsf{ACCESS}(n) \\ \mathcal{C}(\lambda.t) & = & \mathsf{GRAB};\ \mathcal{C}(t) \\ \mathcal{C}(t\ u) & = & \mathsf{PUSH}(\mathcal{C}(u));\ \mathcal{C}(t) \end{array}$$

- ACCESS(n): access the $n$-th thunk of the environment, and start evaluating it (in the SECD, the accessed element was rather pushed on the stack).
- PUSH(c): make a thunk with the code $c$ and the currrent environment, and push it on the stack.
- GRAB: takes a thunk on the top of the stack, and move it to the environment.

Execution of a term $(\lambda.\lambda.t)\ a_1\ a_2\ a_3$ proceeds by:

1. pushing thunks for the arguments $a_3$, $a_2$ and $a_1$ on the stack,
2. then applying the function $\lambda.\lambda.t$ by transferring the thunks for $a_1$ and $a_2$ from the stack to the environment,
3. finally executing the code of $t$, evaluating the thunks for $a_1$ or $a_2$ when the corresponding de Bruijn indices are accessed (and possibly transferring the thunk for $a_3$ to the environment if a new $\lambda$ is encountered).

Machine steps. We write $\langle c, e \rangle$ for the thunk made with code $c$ and environment $e$.

| State before | | | State after | | | |
|---|---|---|---|---|---|---|
| code | env | stack | code | env | stack | |
| ACCESS($n$); $c$ | $e$ | $s$ | $c'$ | $e'$ | $s$ | if $e[n] = \langle c', e' \rangle$ |
| PUSH($c'$); $c$ | $e$ | $s$ | $c$ | $e$ | $\langle c', e \rangle \cdot s$ | |
| GRAB; $c$ | $e$ | $\langle c', e' \rangle \cdot s$ | $c$ | $\langle c', e' \rangle \cdot e$ | $s$ | |

The execution of this machine can be related step by step to small-step reduction of a $\lambda$-calculus extended with an explicit representation of the environment (*$\lambda$-calculus with explicit substitutions*).