The λ-calculus, typed and untyped

Lecture 5: Programming with the untyped λ-calculus

Giulio Guerrieri

✉ giulio.guerrieri@uibk.ac.at https://cl-teaching.uibk.ac.at/26w/lc
Institut für Informatik, Universität Innsbruck

Innsbruck, Austria – 13 November 2026

Outline

1. Programming with the untyped λ-calculus
2. Conclusion, exercises and bibliography

Outline

1. Programming with the untyped λ-calculus
2. Conclusion, exercises and bibliography

Encoding Booleans

Goal. Encode propositional classical logic in the untyped λ-calculus.

We choose (arbitrarily) two terms to represents true $\top$ and false $\bot$.

$$\underline{\top} = \lambda x.\lambda y.x \qquad \underline{\bot} = \lambda x.\lambda y.y$$

Rmk. For every s, t, we have $\underline{\top}\, s\, t \to_{h\beta} (\lambda y.s)t \to_{h\beta} s$ and $\underline{\bot}\, s\, t \to_{h\beta} (\lambda y.y)s \to_{h\beta} t$.

1. We look for a term to encode the NOT: $\underline{not}\, \underline{\top} \to_\beta^* \underline{\bot}$ and $\underline{not}\, \underline{\bot} \to_\beta^* \underline{\top}$.
$$\underline{not} = \lambda p.p\underline{\bot}\,\underline{\top}$$
2. To encode the AND: $\underline{and}\, s\, t \to_\beta^* \underline{\top}$ if $s = t = \underline{\top}$, but $\underline{and}\, s\, t \to_\beta^* \underline{\bot}$ if $s = \underline{\bot}$ or $t = \underline{\bot}$.
$$\underline{and} = \lambda p.\lambda q.pqp$$
3. To encode the OR: $\underline{or}\, s\, t \to_\beta^* \underline{\bot}$ if $s = t = \underline{\bot}$, but $\underline{or}\, s\, t \to_\beta^* \underline{\bot}$ if $s = \underline{\top}$ or $t = \underline{\top}$.
$$\underline{or} = \lambda p.\lambda q.ppq$$
4. To encode the IF-THEN-ELSE: $\underline{if}\, r\, s\, t \to_\beta^* s$ if $r = \underline{\top}$ and $\underline{if}\, r\, s\, t \to_\beta^* t$ if $r = \underline{\bot}$.
$$\underline{if} = \lambda p.\lambda a.\lambda b.pab$$

How to check that the encoding of Booleans is sound?

1. For each operator, apply its encoding to the encoding of each element of its domain,
2. Check that the returned term is the encoding of the expected value.

The domain of Booleans is finite (only 2 values) $\rightsquigarrow$ Checking all cases is doable.

$$\underline{not}\,\underline{\top} = (\lambda p.p\underline{\bot}\,\underline{\top})\underline{\top} \to_\beta \underline{\top}\,\underline{\bot}\,\underline{\top} \to_\beta^* \underline{\bot}$$
$$\underline{not}\,\underline{\bot} = (\lambda p.p\underline{\bot}\,\underline{\top})\underline{\bot} \to_\beta \underline{\bot}\,\underline{\bot}\,\underline{\top} \to_\beta^* \underline{\top}$$

$$\underline{and}\,\underline{\top}\,\underline{\top} = (\lambda p.\lambda q.pqp)\underline{\top}\,\underline{\top} \to_\beta (\lambda q.\underline{\top}\,q\,\underline{\top})\underline{\top} \to_\beta \underline{\top}\,\underline{\top}\,\underline{\top} \to_\beta^* \underline{\top}$$
$$\underline{and}\,\underline{\top}\,\underline{\bot} = (\lambda p.\lambda q.pqp)\underline{\top}\,\underline{\bot} \to_\beta (\lambda q.\underline{\top}\,q\,\underline{\top})\underline{\bot} \to_\beta \underline{\top}\,\underline{\bot}\,\underline{\top} \to_\beta^* \underline{\bot}$$
$$\underline{and}\,\underline{\bot}\,s = (\lambda p.\lambda q.pqp)\underline{\bot}\,s \to_\beta (\lambda q.\underline{\bot}\,q\,\underline{\bot})s \to_\beta \underline{\bot}\,s\,\underline{\bot} \to_\beta^* \underline{\bot}$$

$$\underline{or}\,\underline{\bot}\,\underline{\bot} = (\lambda p.\lambda q.ppq)\underline{\bot}\,\underline{\bot} \to_\beta (\lambda q.\underline{\bot}\,\underline{\bot}\,q)\underline{\bot} \to_\beta \underline{\bot}\,\underline{\bot}\,\underline{\bot} \to_\beta^* \underline{\bot}$$
$$\underline{or}\,\underline{\top}\,s = (\lambda p.\lambda q.ppq)\underline{\top}\,s \to_\beta (\lambda q.\underline{\top}\,\underline{\top}\,q)s \to_\beta \underline{\top}\,\underline{\top}\,s \to_\beta^* \underline{\top}$$
$$\underline{or}\,\underline{\bot}\,\underline{\top} = (\lambda p.\lambda q.ppq)\underline{\bot}\,\underline{\top} \to_\beta (\lambda q.\underline{\bot}\,\underline{\bot}q)\underline{\top} \to_\beta \underline{\bot}\,\underline{\bot}\,\underline{\top} \to_\beta^* \underline{\top}$$

$$\underline{if}\,\underline{\top}\,s\,u = (\lambda p.\lambda a.\lambda b.pab)\underline{\top}\,s\,u \to_\beta (\lambda a.\lambda b.\underline{\top}ab)su \to_\beta (\lambda b.\underline{\top}\,s\,b)u \to_\beta \underline{\top}\,s\,u \to_\beta^* s$$
$$\underline{if}\,\underline{\bot}\,s\,u = (\lambda p.\lambda a.\lambda b.pab)\underline{\bot}\,s\,u \to_\beta (\lambda a.\lambda b.\underline{\bot}ab)su \to_\beta (\lambda b.\underline{\bot}\,s\,b)u \to_\beta \underline{\bot}\,s\,u \to_\beta^* u$$

Encoding arithmetic

Goal. Encode the arithmetic in the untyped λ-calculus.

We choose a term $\underline{n}$ to represents any $n \in \mathbb{N}$ (Church numeral).

$$\underline{n} = \lambda f.\lambda x.f^n x = \lambda f.\lambda x.\underbrace{f(f\ldots(f}_{n \text{ times } f}\,x)\ldots) \qquad \text{(in particular, } \underline{0} = \lambda f.\lambda x.x)$$

Rmk. For any s, t, we have $\underline{n}\,s\,t \to_{h\beta} (\lambda x.s^n x)t \to_{h\beta} s^n t = \overbrace{s(s\ldots(s}^{n \text{ times } s}\,t)\ldots)$ (n-iterator).

1. We look for a term to encode the successor: $\underline{succ}\,\underline{n} \to_\beta^* \underline{n+1}$.
$$\underline{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$
2. To encode the addition: $\underline{add}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m+m}$.
$$\underline{add} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$
3. To encode the multiplication: $\underline{mult}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m \times n}$.
$$\underline{mult} = \lambda m.\lambda n.\lambda f.m(nf)$$
4. To encode the exponentiation: $\underline{pow}\,\underline{m}\,\underline{n} \to_\beta^* \underline{m^n}$.
$$\underline{pow} = \lambda m.\lambda n.nm$$

How to check that the encoding of arithmetic is sound?

$$\underline{add}\,\underline{2}\,\underline{3} = (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))\underline{2}\,\underline{3} \to_\beta (\lambda n.\lambda f.\lambda x.\underline{2}\,f(nfx))\underline{3} \to_\beta \lambda f.\lambda x.\underline{2}\,f(\underline{3}fx)$$
$$\to_\beta^* \lambda f.\lambda x.\underline{2}\,f(f^3x) \to_\beta^* \lambda f.\lambda x.f^2(f^3x) = \lambda f.\lambda x.f^5x = \underline{5}$$

The domain of natural numbers is infinite $\rightsquigarrow$ Checking all cases is impossible.

Idea. Prove it for arbitrary $m, n \in \mathbb{N}$! Sometimes it could be helpful to use induction.

$$\underline{add}\,\underline{m}\,\underline{n} = (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))\underline{m}\,\underline{n} \to_\beta (\lambda n.\lambda f.\lambda x.\underline{m}\,f(nfx))\underline{n} \to_\beta \lambda f.\lambda x.\underline{m}\,f(\underline{n}\,fx)$$
$$\to_\beta^* \lambda f.\lambda x.\underline{m}\,f(f^n x) \to_\beta^* \lambda f.\lambda x.f^m(f^n x) = \lambda f.\lambda x.f^{m+n}x = \underline{m+n}$$

$$\underline{mult}\,\underline{m}\,\underline{n} = (\lambda m.\lambda n.\lambda f.m(nf))\underline{m}\,\underline{n} \to_\beta (\lambda n.\lambda f.\underline{m}(nf))\underline{n} \to_\beta \lambda f.\underline{m}(\underline{n}\,f)$$
$$\to_\beta \lambda f.\underline{m}\,(\lambda z.f^n z) \to_\beta (\lambda x.(\lambda z.f^n z)^m x) \to_\beta^* \lambda f.\lambda x.f^{m\cdot n}x = \underline{m \cdot n}$$

where we prove that $(\lambda z.f^n z)^m x \to_\beta^* f^{m\cdot n}x$ by induction on $m \in \mathbb{N}$:

$$(\lambda z.f^n z)^0 x = x = f^{0\cdot n}x$$
$$(\lambda z.f^n z)^{m+1}x = (\lambda z.f^n z)((\lambda z.f^n z)^m x) \overset{IH}{\to}_\beta^* (\lambda z.f^n z)(f^{m\cdot n}x) \to_\beta f^n(f^{m\cdot n}x) = (f^{(m+1)\cdot n}x)$$

More about encoding arithmetic: recursion

We can encode the functions: $iszero\colon \mathbb{N} \to \{\bot, \top\}$ testing if a natural number is 0 or not, and the predecessor $pred\colon \mathbb{N} \to \mathbb{N}$ such that $pred(0) = 0$ and $pred(n+1) = n$.

$$\underline{iszero} = \lambda n.n(\lambda x.\underline{\bot})\underline{\top} \qquad \underline{iszero}\,\underline{n} \to_\beta^* \begin{cases} \underline{\top} & \text{if } n = 0 \\ \underline{\bot} & \text{otherwise.} \end{cases} \qquad \underline{pred} = \ldots$$

Question. How can the λ-calculus represent the factorial (typical recursive function)?

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{otherwise.} \end{cases}$$

Let us rewrite the definition in a λ-calculus-like style, using IF-THEN-ELSE and $\underline{mult}$:

$\underline{fact}$ should satisfies the equation: $\underline{fact}\,\underline{n} = \underline{if}\,(\underline{iszero}\,\underline{n})\,\underline{1}\,(\underline{mult}\,\underline{n}\,(\underline{fact}\,(\underline{pred}\,\underline{n})))$ $\underline{fact}$ should sat

$F := \lambda f.\lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(f\,(\underline{pred}\,n)))$ Y is a fixpoint combinator

$\underline{fact} := YF \to_\beta^* F(YF) = F\,\underline{fact} \to_\beta \lambda n.\underline{if}\,(\underline{iszero}\,n)\,\underline{1}\,(\underline{mult}\,n\,(\underline{fact}\,(\underline{pred}\,n)))$

The power of fixpoint combinators to solve recursive reductions/equations

Def. A fixpoint combinator is a term Y such that $Yt \to_\beta^* t(Yt)$ (see Lecture 4).
A weak fixpoint combinator is a term Y such that $Yt =_\beta t(Yt)$.

Rmk. Every fixpoint combinator is a weak fixpoint combinator, but the converse fails.

Ex. $\Xi = \lambda f.B_f B_f$ where $B_f = \lambda x.f(xx)$ is a weak fixpoint combinator but not a fixpoint combinator. $\Theta = AA$ where $A = \lambda x.\lambda f.f(xxf)$ is a fixpoint combinator.

Consider the reduction below on the left or the equation below on the right:

$$t \to_\beta^* \ldots t \ldots \qquad\qquad t =_\beta \ldots t \ldots \tag{1}$$

or more formally

$$t \to_\beta^* s\{t/x\} \qquad\qquad t =_\beta s\{t/x\} \tag{2}$$

where on either side of the sign $\to_\beta$ or $=_\beta$, an unknown t occurs. Does such a t exist?

Theorem

For every term s, a solution t to the reduction/equation (2) exists.

Proof. Let $S = \lambda x.s$ and $t = YS$, where Y is a fixpoint (resp. weak fixpoint) combinator. Then $t \to_\beta^* S(YS)$ (resp. $t =_\beta S(YS)$) with $S(YS) = (\lambda x.s)t \to_\beta s\{t/x\}$. □

Encoding pairs and selectors

A pair packs two values a and b into an ordered object $\langle a, b\rangle$. A selector is a function selecting the first or second element of a pair: $fst\langle a, b\rangle = a$ and $snd\langle a, b\rangle = b$.

The pair constructor *pair* (also denoted by $\langle\cdot,\cdot\rangle$) is represented in the λ-calculus by

$$\text{pair} = \lambda a.\lambda b.\lambda f.fab \qquad (\text{so } \langle s, t\rangle = \text{pair}\, s\, t \to_\beta^* \lambda f.fst)$$

The selectors *fst* and *snd* are represented in the λ-calculus by:

$$\text{fst} = \lambda f.f\underline{\top} \qquad \text{snd} = \lambda f.f\underline{\bot} \qquad \text{with } \underline{\top} = \lambda x.\lambda y.x \text{ and } \underline{\bot} = \lambda x.\lambda y.y.$$

Indeed, for every terms s and t,

$$\text{fst}\,\langle s, t\rangle = (\lambda f.f\underline{\top})(\text{pair}\, s\, t) \to_\beta^* (\lambda f.f\, s\, t)\underline{\top} \to_\beta (\lambda x.\lambda y.x)s\, t \to_\beta (\lambda y.s)t \to_\beta s$$
$$\text{snd}\,\langle s, t\rangle = (\lambda f.f\underline{\bot})(\text{pair}\, s\, t) \to_\beta^* (\lambda f.f\, s\, t)\underline{\bot} \to_\beta (\lambda x.\lambda y.y)s\, t \to_\beta (\lambda y.y)t \to_\beta t$$

Encoding lists

Given the terms $t_1, \ldots, t_k$, the list $[t_1, \ldots, t_k]$ can be encoded in the λ-calculus as:

$$[t_1, \ldots, t_k] = \lambda c.\lambda n.ct_1(\ldots(ct_k n)\ldots) \qquad (\text{in particular for } k = 0,\ [\,] = \lambda c.\lambda n.n)$$

Ex. $[\underline{1}, \underline{2}, \underline{3}, \underline{4}] = \lambda c.\lambda n.c\,\underline{1}\,(c\,\underline{2}\,(c\,\underline{3}\,(c\,\underline{4}\,n)))$ where $\underline{n}$ is the Church numeral of $n \in \mathbb{N}$.

The usual operations on lists can also be encoded in the λ-calculus.

1. constructor adding a term in front of a list: $\text{cons}\, u\, [t_1, \ldots, t_k] \to_\beta^* [u, t_1 \ldots, t_k]$
$$\text{cons} = \lambda x.\lambda \ell.\lambda c.\lambda n.cx(\ell cn)$$
2. head extracting the first term from a non-empty list: $\text{head}\, [t_0, \ldots, t_k] \to_\beta^* t_0$
$$\text{head} = \lambda \ell.\ell(\lambda x.\lambda y.x)(\lambda z.z)$$
3. tail removing the first term from a non-empty list: $\text{tail}\, [t_0, t_1, \ldots, t_k] \to_\beta^* [t_1, \ldots, t_k]$
$$\text{tail} = \lambda \ell.\lambda c.\lambda n.\ell(\lambda h.\lambda r.\lambda g.gh(rc))(\lambda g.n)(\lambda x.\lambda y.y)$$
4. map applying a term to each element of a list: $\text{map}\, s\, [t_0, \ldots, t_k] \to_\beta^* [st_1, \ldots, st_k]$
$$\text{map} = \lambda \ell.\lambda c.\lambda n.\ell(\lambda h.\lambda r.c(fr))n$$
5. foldr right folding a function on a list: $\text{foldr}\, s\, u\, [t_0, \ldots, t_k] \to_\beta^* st_1(\ldots(st_k u)\ldots)$
$$\text{foldr} = \lambda c.\lambda n.\lambda \ell.\ell cn$$

The untyped λ-calculus is Turing-complete!

Def. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial. A term Φ represents f when, for all $k_1, \ldots, k_n \in \mathbb{N}$:

1. if $f(k_1, \ldots, k_n)$ is undefined, then $\Phi\, \underline{k_1} \ldots \underline{k_n}$ is not $h\beta$-normalizing;
2. if $f(k_1, \ldots, k_n) = k \in \mathbb{N}$, then $\Phi\, \underline{k_1} \ldots \underline{k_n} \to_\beta^* \underline{k}$.

Theorem (Representability)

Every partial recursive function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is representable by a term in the λ-calculus.

Rmk. According to Church's thesis, the λ-calculus can represent everything is computable.

Rmk. If Φ represents a partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, then Φ could have whatever behavior when applied to arguments $t_1, \ldots, t_k$ that are not Church numerals.

Rmk. In Point 1 of the definition, $h\beta$-normalizing can be replaced by β-normalizing.

The λ-calculus as a programming language

λ-calculus = kernel of all functional programming languages (Haskell, OCaml, Lisp, ...).

- An abstraction $\lambda x.t$ is an anonymous function `fun x -> t` in OCaml.
- A application tu (resp. variable x) is an application `tu` (resp. variable `x`) in OCaml.

OCaml	λ-calculus
`(fun x -> x + 3)7`	$(\lambda x.\underline{add}\, x\, \underline{3})\, \underline{7}$
$\Downarrow$	$\downarrow_\beta$
`7 + 3`	$\underline{add}\, \underline{7}\, \underline{3}$
$\Downarrow$	$*\downarrow_\beta$
`10`	$\underline{10}$

- OCaml imposes eager evaluation: evaluate arguments before applying the function.
- In the λ-calculus the order of evaluation is irrelevant (because of confluence).

Outline

1 Programming with the untyped λ-calculus

2 Conclusion, exercises and bibliography

What we have learned today?

1. How to encode arithmetic and propositional classical logic in the untyped λ-calculus.
2. Use of fixpoint combinators to solve recursive equations in the untyped λ-calculus.
3. How to encode data structures such as pairs and lists in the untyped λ-calculus.
4. Claim that all partial recursive functions are representable in the untyped λ-calculus.

Questions?



Non-assessed exercises (assessed exercises are available on: click here)

1. Prove that $\underline{succ}\, \underline{n} \to_\beta^* \underline{n+1}$ for all $n \in \mathbb{N}$, and $\underline{pow}\, \underline{m}\, \underline{n} \to_\beta^* \underline{m^n}$ for all $m, n \in \mathbb{N}$.
2. Prove that the term $\underline{pred} = \lambda n.\lambda f.\lambda x.n(\lambda r\lambda i.i(rf))(\lambda f.x)\lambda z.z$ represents the function *pred* on natural numbers defined by $pred(0) = 0$ and $pred(n+1) = n$.
3. Using the term $\underline{pred}$, define a term $\underline{minus}$ that represents the function *minus* on $\mathbb{N}$ defined by $minus(m, n) = 0$ if $m < n$, and $minus(m, n) = m - n$ otherwise.
4. Define a term $\underline{add}$ that represents the addition of natural numbers starting from its inductive definition below (*Hint:* Use the fixpoint combinator Θ, $\underline{pred}$, $\underline{iszero}$).

$$m + n = \begin{cases} m & \text{if } n = 0 \\ m + (n-1) & \text{otherwise;} \end{cases}$$

5. Define a term $\underline{mul}$ that represents the multiplication of natural numbers starting from its inductive definition below (*Hint:* Use fixpoint combinator Θ, $\underline{pred}$, $\underline{iszero}$).

$$m \times n = \begin{cases} 0 & \text{if } n = 0 \\ m + m \times (n-1) & \text{otherwise.} \end{cases}$$

6. Verify that $\Xi = \lambda f.B_f B_f$ where $B_f = \lambda x.f(xx)$ is a weak fixpoint combinator but not a fixpoint combinator.
7. Verify that the terms cons, head, tail, map, foldr on p. 11 behave as expected.
8. Execute the term $[\underline{1}, \underline{3}, \underline{5}]\, \underline{add}\, \underline{0}$ until its β-normal form. What is the result?
9. Write a term that when applied to a list of Church numerals, returns their product.

Bibliography

- For more about the untyped λ-calculus:


Jean-Louis Krivine. *Lambda-Calculus. Types and Models*. Ellis Horwood. 1990. [Chapters 1-2] `https://www.irif.fr/~krivine/articles/Lambda.pdf`


Peter Selinger. *Lecture Notes on the Lambda Calculus*. vol. 0804, Department of Mathematics and Statistics, University of Ottawa. 2008 [Chapters 2-3] `http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf`


Henk P. Barendregt. *The Lambda-Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103, North Holland, 1984. [Chapters 2-3, 6, 8]