Even though the pure $\lambda$-calculus consists only of $\lambda$-terms, we can represent and manipulate common data objects like integers, Boolean values, lists, and trees. All these things can be encoded as $\lambda$-terms.

# 1 Encoding Common Datatypes

## 1.1 Booleans

The Booleans are the easiest to encode, so let us start with them. We would like to define $\lambda$-terms to represent the Boolean constants $\mathsf{true}$ and $\mathsf{false}$ and the usual Boolean operators $\Rightarrow$ (if-then), $\wedge$ (and), $\vee$ (or), and $\neg$ (not) so that they behave in the expected way. There are many reasonable encodings. One good one is to define $\mathsf{true}$ and $\mathsf{false}$ by:

$$\mathsf{true} \triangleq \lambda xy.\, x \qquad\qquad \mathsf{false} \triangleq \lambda xy.\, y.$$

Now we would like to define a conditional test $\mathsf{if}$. We would like $\mathsf{if}$ to take three arguments $b, t, f$, where $b$ is a Boolean value (either $\mathsf{true}$ or $\mathsf{false}$) and $t, f$ are arbitrary $\lambda$-terms. The function should return $t$ if $b = \mathsf{true}$ and $f$ if $b = \mathsf{false}$.

$$\mathsf{if} \quad = \quad \lambda btf.\, \begin{cases} t, & \text{if } b = \mathsf{true}, \\ f, & \text{if } b = \mathsf{false}. \end{cases}$$

Now the reason for defining $\mathsf{true}$ and $\mathsf{false}$ the way we did becomes clear. Since $\mathsf{true}\, t\, f \xrightarrow{1} t$ and $\mathsf{false}\, t\, f \xrightarrow{1} f$, all $\mathsf{if}$ has to do is apply its Boolean argument to the other two arguments:

$$\mathsf{if} \quad \triangleq \quad \lambda btf.\, btf$$

The other Boolean operators can be defined from $\mathsf{if}$:

$$\mathsf{and} \triangleq \lambda b_1 b_2.\, \mathsf{if}\, b_1\, b_2\, \mathsf{false} \qquad \mathsf{or} \triangleq \lambda b_1 b_2.\, \mathsf{if}\, b_1\, \mathsf{true}\, b_2 \qquad \mathsf{not} \triangleq \lambda b_1.\, \mathsf{if}\, b_1\, \mathsf{false}\, \mathsf{true}$$

Whereas these operators work correctly when given Boolean values as we have defined them, all bets are off if they are applied to any other $\lambda$-term. There is no guarantee of any kind of reasonable behavior. Basically, with the untyped $\lambda$-calculus, it is *garbage in, garbage out.*

## 1.2 Natural Numbers

We will encode natural numbers $\mathbb{N}$ using *Church numerals*. This is the same encoding that Alonzo Church used, although there are other reasonable encodings. The Church numeral for the number $n \in \mathbb{N}$ is denoted

$\overline{n}$. It is the $\lambda$-term $\lambda fx.\, f^n\, x$, where $f^n$ denotes the $n$-fold composition of $f$ with itself:

$$\begin{array}{rcll}
\overline{0} & \triangleq & \lambda fx.\, f^0 x & = \lambda fx.\, x \\
\overline{1} & \triangleq & \lambda fx.\, f^1 x & = \lambda fx.\, fx \\
\overline{2} & \triangleq & \lambda fx.\, f^2 x & = \lambda fx.\, f(fx) \\
\overline{3} & \triangleq & \lambda fx.\, f^3 x & = \lambda fx.\, f(f(fx)) \\
 & \vdots & & \\
\overline{n} & \triangleq & \lambda fx.\, f^n x & = \lambda fx.\, \underbrace{f(f(\ldots(f}_{n}\, x)\ldots))
\end{array}$$

We can define the successor function $\mathsf{succ}$ as

$$\mathsf{succ} \triangleq \lambda nfx.\, f\,(n\, f\, x).$$

That is, $\mathsf{succ}$ on input $\overline{n}$ returns a function that takes a function $f$ as input, applies $\overline{n}$ to it to get the $n$-fold composition of $f$ with itself, then composes that with one more $f$ to get the $(n+1)$-fold composition of $f$ with itself. Then

$$\begin{array}{rcl}
\mathsf{succ}\,\overline{n} & = & (\lambda nfx.\, f(nfx))\,\overline{n} \\
 & \xrightarrow{1} & \lambda fx.\, f(\overline{n}fx) \\
 & \xrightarrow{1} & \lambda fx.\, f(f^n x) \\
 & = & \lambda fx.\, f^{n+1} x \\
 & = & \overline{n+1}.
\end{array}$$

We can perform basic arithmetic with Church numerals. For addition, we might define

$$\mathsf{add} \triangleq \lambda mnfx.\, mf(nfx).$$

On input $\overline{m}$ and $\overline{n}$, this function returns

$$\begin{array}{rcl}
(\lambda mnfx.\, mf(nfx))\,\overline{m}\,\overline{n} & \xrightarrow{1} & \lambda fx.\, \overline{m}f(\overline{n}fx) \\
 & \xrightarrow{1} & \lambda fx.\, f^m(f^n x) \\
 & = & \lambda fx.\, f^{m+n} x \\
 & = & \overline{m+n}.
\end{array}$$

Here we are composing $f^m$ with $f^n$ to get $f^{m+n}$.

Alternatively, recall that Church numerals act on a function to apply that function repeatedly, and addition can be viewed as repeated application of the successor function, so we could define

$$\mathsf{add} \triangleq \lambda mn.\, m\,\mathsf{succ}\, n.$$

Similarly, multiplication is just iterated addition, and exponentiation is iterated multiplication:

$$\mathsf{mul} \triangleq \lambda mn.\, m\,(\mathsf{add}\, n)\,\overline{0} \qquad \mathsf{exp} \triangleq \lambda mn.\, m\,(\mathsf{mul}\, n)\,\overline{1}.$$

### 1.3 Pairing and Projections

Logic and arithmetic are good places to start, but we still are lacking any useful data structures. For example, consider ordered pairs. It would be nice to have a pairing function pair with projections first and second that obeyed the following equational specifications:

$$\mathsf{first}\,(\mathsf{pair}\,e_1\,e_2) \;=\; e_1 \qquad \mathsf{second}\,(\mathsf{pair}\,e_1\,e_2) \;=\; e_2 \qquad \mathsf{pair}\,(\mathsf{first}\,p)\,(\mathsf{second}\,p) \;=\; p,$$

provided $p$ is a pair. We can take a hint from if. Recall that if selects one of its two branch options depending on its Boolean argument. pair can do something similar, wrapping its two arguments for later extraction by some function $f$:

$$\mathsf{pair} \;\stackrel{\triangle}{=}\; \lambda abf\,.\,fab.$$

Thus $\mathsf{pair}\,e_1\,e_2 \;\to\; \lambda f\,.\,fe_1e_2$. To get $e_1$ back out, we can just apply this to true: $(\lambda f\,.\,fe_1e_2)\,\mathsf{true} \;\to\; \mathsf{true}\,e_1\,e_2 \to e_1$, and similarly applying it to false extracts $e_2$. Thus we can define

$$\mathsf{first} \;\stackrel{\triangle}{=}\; \lambda p\,.\,p\,\mathsf{true} \qquad \mathsf{second} \;\stackrel{\triangle}{=}\; \lambda p\,.\,p\,\mathsf{false}.$$

Again, if $p$ is not a term of the form $\mathsf{pair}\,a\,b$, expect the unexpected.

### 1.4 Lists

One can define lists $[x_1; \ldots; x_n]$ and list operators corresponding to the OCaml `::`, `List.hd`, and `List.tl` in the $\lambda$-calculus. We leave these constructions as exercises.

### 1.5 Local Variables

One feature that seems to be missing is the ability to declare local variables. For example, in OCaml, we can introduce a new local variable with the let expression:

$$\mathsf{let}\;x = e_1\;\mathsf{in}\;e_2$$

Intuitively, we expect this expression to evaluate $e_1$ to some value $v$ and then to replace occurrences of $x$ inside $e_2$ with $v$. In other words, it should evaluate to $e_2\{v/x\}$. But we can construct a $\lambda$-term that behaves the same way:

$$(\lambda x\,.\,e_2)\,e_1 \;\to\; (\lambda x\,.\,e_2)\,v \;\stackrel{1}{\to}\; e_2\{v/x\}.$$

We can thus view a let expression as syntactic sugar for an application of a $\lambda$-abstraction.

## References


[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[2] Henk P. Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. and Comput.*, 207(5):559–582, 2009.

[3] James Gosling, Bill Joy, Jr. Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* Prentice Hall, 3rd edition, 2005.

[4] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[5] Jean-Baptiste Jeannin. Capsules and closures. *Electron. Notes Theor. Comput. Sci.*, 276:191–213, September 2011.

[6] Jean-Baptiste Jeannin and Dexter Kozen. Capsules and separation. In Nachum Dershowitz, editor, *Proc. 27th ACM/IEEE Symp. Logic in Computer Science (LICS'12)*, pages 425–430, Dubrovnik, Croatia, June 2012. IEEE.

[7] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Proc. Conf. Descriptional Complexity of Formal Systems (DCFS 2012)*, volume 7386 of *Lecture Notes in Computer Science*, pages 1–19, Braga, Portugal, July 2012. Springer.

[8] J. W. Klop and R. C. de Vrijer. Infinitary normalization. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

[9] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[10] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.

[11] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of programming languages I*, pages 173–185. ACM, 1981.

[12] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[13] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[14] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[15] Robert Pollack. Polishing up the Tait–Martin-Löf proof of the Church–Rosser theorem. In *Proc. De Wintermöte '95*. Department of Computing Science, Chalmers University, Göteborg, Sweden, January 1995.

[16] Masako Takahashi. Parallel reductions in $\lambda$-calculus (revised version). *Information and Computation*, 118(1):120–127, April 1995.

[17] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[18] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, August 1992.

[19] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.