Gems of Corrado Böhm

Henk Barendregt

November 6, 2019

To the memory of Corrado Böhm (1923-2017)

Abstract

The main scientific heritage of Corrado Böhm is about computing, concerning concrete algorithms as well as models of computability. Discussed will be the following. 1. A compiler that can compile itself. 2. Structured programming, eliminating the 'goto' statement. 3. Functional programming and an early implementation. 4. Separability in λ-calculus. 5. Compiling combinators without parsing. 6. Self-evaluation in λ-calculus.

Introduction

As a tribute to Corrado Böhm this paper explains six important results of his and also discusses some of their later developments. Most of the papers are written by Böhm with co-authors. The result on elimination of the `goto`, Section 2, is written by Giuseppe Jacopini alone in the joint paper with Böhm [17], but one may assume that Böhm as supervisor had influenced the research involved, and therefore this result is included here. This paper is written such that computer science freshmen can read and understand it.

1. Self-compilation

In his PhD thesis [11] at the ETH Zürich, Corrado Böhm constructed one of the first higher programming languages L together with a compiler for it. The compiler has the particular quality that it is written in the language L itself. This sounds like magic, but it is not: if a programming language is capable of expressing any computational process, then it should also be able to 'understand itself' (i.e. perform the computational task to translate it into machine language). Later this property gave rise to 'bootstrapping': dramatically increasing efficiency and reliability of computer programs, that seems as impossible as to pull oneself over a fence by pulling one's bootstraps[1]. This gave rise to the term 'booting a computer'.

[1] In Europe the hyperbole for impossibility is the story of Baron (von) Münchhausen, who could get himself (and the horse on which he was seated) out of a swamp by pulling up his own hair.

1.1. Algorithms, computers, and imperative programming

An *algorithm* is a recipe to compute an output from a given input. Executing such a recipe basically consists in putting down pebbles[2] in a fixed array of boxes and 'replacing' these pebbles step by step. That is, a pebble may be moved from one box to another one, be taken away, or new ones may be added. Such a process is called a *calculation* or *computation*. As Turing [54] has shown, all computational tasks, like "What is the square of 29?", "Put the following list of words in alphabetical order", or "What does Wikipedia say about the concept 'bootstrap'?", can be put in the format of shuffling pebbles in boxes.

This view on computing holds for computations on an abacus, but also for programmed *computers*. A computer M is a, usually electronic, device with memory, that performs computations. The pebbles are represented in this memory and the shuffling is done by making stepwise changes to them. A simple conceptual computer is the Turing Machine (TM). It consists of an infinite[3] tape of discrete cells that can be numbered by the integers $\mathbb{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$. At every moment in the computation on only a finite number of these cells information is written, either a 1 or nothing, denoted by 0: the original TM was a 0-bit[4] machine. The machine can be in one of a finite number of *states*. There is a read/write (R/W) head positioned on one of the cells of the tape. Depending on the symbol a that is read, and the present state s, one of the following three actions is performed: a (possibly different) symbol a' is written on the cell under the R/W-head, a (possibly different) state s' is assumed, and finally the head moves $\{R, L, N\}$ (R: one position to the right, L: one position to the left, N: no moving). Each Turing Machine is determined by a finite table consisting of 5-tuples like $\langle a, s; a', s', \{R, L, N\}\rangle$ that determine the changes.

Turing showed that there exists a particular kind of machine, called a *universal machine* $\mathcal{U}$, that suffices to make arbitrary computations. Such a $\mathcal{U}$ is conceptually easy. The set of 5-tuples of a particular machine $\mathcal{M}$ is presented as a table $T_{\mathcal{M}}$ 'in its silicon'. A universal machine $\mathcal{U}$ that imitates $\mathcal{M}$, needs in coded form this table $T_{\mathcal{M}}$, including the collection of all states of $\mathcal{M}$ (that may be more extensive than that of $\mathcal{U}$) and the present state of $\mathcal{M}$, stored in a dedicated part of the memory as the program (nowdays known as the 'app') for $\mathcal{M}$. The instruction table $T_{\mathcal{U}}$ of $\mathcal{U}$ stipulates that it 1. has to look in $\mathcal{T}_{\mathcal{M}}$ in order to see what is the present state of $\mathcal{M}$, and to know what to do next; and 2. to do this. The possibility of a universal machine provides a model of computation in which a single machine $\mathcal{M}$, using programming language $M = L_{\mathcal{M}}$, can perform any computational job. The nature of the actions of Turing machines, described in their action tables, is rather imperative: overwrite information, change state, move. For this reason the resulting computational model is called *imperative programming*.

In this paper we will consider a fixed universal machine $\mathcal{M}$. Around 1950, when

[2]The word 'pebble' in Latin is 'calculus'.

[3]Actual computers have only a finite amount of memory. Turing apparently didn't want to be technology dependent and conceived the Turing Machine with an idealized memory of infinitely many cells. In actual computers there is only a bounded amount of information that can be stored.

[4]In 0-bit machines counting happens in the 2^0-ary, i.e. unary, system. In modern computers the cells are replaced by registers that contain a sequence of 64 or more bits that can be read or overwritten in parallel; moreover, the registers do not need to be looked up linearly, like on the tape of the TM, but there is fast access to each of them; one speaks of 'random access memory' (RAM).

Corrado Böhm worked on his PhD, computers were rare. Indeed, in 1954, in a country like the Netherlands there were only three computers (at the Mathematical Center, the Royal Meteorological Institute, and the National Phone Company) and no more were deemed to be necessary! Nowadays (2019) a standard car has on board at least > 100 (universal) computers.

A program in a given language M for $\mathcal{M}$ consists of a sequence of *statements* in M that the machine 'understands': it performs intended changes on data represented in the memory of $\mathcal{M}$. Such programs are denoted by $p = p^M$, the superscript indicating that the program is written in the language M.

1.1. Definition. (i) There is a non-specified set D (for data) consisting of the intended objects on which computations take place.

(ii) The process of running program p^M on input x in D is denoted by $\{p^M\}(x)$[5]. If this process terminates with end result y (the output, again in D), then we write

$$\{p^M\}(x) \twoheadrightarrow y.$$

(iii) It may be the case that $\{p^M\}(x)$ doesn't terminate. Then there is no output, and we write $\{p^M\}(x)\uparrow$.

(iv) The (operational) *semantics* of p^M is the partial map $[\![p^M]\!]: D \rightharpoonup D$ defined as follows.

$$\begin{aligned} [\![p^M]\!](x) &= y, &&\text{if } \{p^M\}(x) \twoheadrightarrow y; \\ &= \uparrow, &&\text{if } \{p^M\}(x)\uparrow. \end{aligned}$$

For $\{\ \}$ and $[\![\ ]\!]$, that depend on M, we often write $\{\ \}_M$, $[\![\ ]\!]_M$, respectively.

The difference between $[\![p^M]\!](x) = y$ and $\{p^M\}(x) \twoheadrightarrow y$ is that the former is a mathematical identity, like $36^2 = 36 \times 36$ that holds by definition, whereas the latter requires a computation, like $36 \times 36 \twoheadrightarrow 1296$. The sign '$\twoheadrightarrow$' indicates that a *computation* has to be performed that takes time, consisting of a sequence of a few or more steps that transform information.

1.2. Proposition. *If* $\{p^M\}(x)$ *terminates, then*

$$\{p^M\}(x) \twoheadrightarrow [\![p^M]\!](x).$$

Proof. By definition. ■

1.2. Programming languages and compilers

A human, having to write a correct and efficient program, better does this in an understandable way, rather than in the form of recipes for shuffling pebbles. One can use a *programming language* L for this, in which computational tasks can be described more intuitively. In [11] an early example of such a language L is constructed.

[5] Compound expressions like $\{\{c\}(p)\}(x)$ make sense and will be used. But an expressions like $\{q\}(\{p\}(x))$ we will avoid, as one is forced to evaluate first the $\{p\}(x)$, which may be undefined; therefore even if $\forall y.\{q\}(y) \twoheadrightarrow 0$, one doesn't always have $\{q\}(\{p\}(x)) \twoheadrightarrow 0$. See [3, Exercise 9.5.13] and [2, 7].

1.3. DEFINITION. (i) A programming language L consists of programs p that describe computations according to (ii).

(ii) L comes with a (denotational) *semantic function* $[\![\ ]\!]_L \colon L \to (D \rightharpoondown D)$. That is for each p^L there is a (possibly partial) function $[\![p^L]\!] \colon D \rightharpoondown D$.

Technically speaking M is also a programming language, the *machine language*, with its denotational semantics $[\![-]\!]_M$, by definition equal to the operational one $\{-\}_M$. By contrast other programming languages are called *higher* programming languages, that are intended to make the construction of programs more easy. When one has a program p^L described in a higher programming language L we want to have machine help from a universal machine to obtain from input x the output $[\![p^L]\!](x)$. We succeed if one can translate p^L in the 'right way' into the machine language M. This translating is called *compiling*.

1.4. DEFINITION. A function $C \colon L_1 \to L_2$, is called a *compiling function* if

$$[\![C(p^{L_1})]\!]_{L_2} = [\![p^{L_1}]\!]_{L_1}.$$

In this paper, we will usually consider only compilers into $L_2 = M$.

1.5. PROPOSITION. *If $C \colon L \to M$ is a compiling function, then*

$$\{C(p^L)\}_M(x) \twoheadrightarrow [\![p^L]\!]_L(x).$$

PROOF. One has by Proposition 1.2 and Definition 1.4

$$\{C(p^L)\}_M(x) \twoheadrightarrow [\![C(p^L)]\!]_M(x) = [\![p^L]\!]_L(x). \blacksquare$$

This shows that an intended computation using a $p^L \in L$, for example executing $[\![p^L]\!](x)$, can in principle be replaced by a computation using a $p^M \in M$, for which there is the support of the machine $\mathcal{M}$. We say: the computational task $[\![p^L]\!](x)$ becomes executable (by $\mathcal{M}$). In modern compilers the translation $L \to M$, is often divided in literally hundreds of steps, using many intermediate languages[6]. For example, the first step is the so called *lexing* that examines where every meaningful unit starts and ends[7]. At the end of the long translation process one arrives at the language M. No need for further translation occurs: in $\mathcal{M}$ the programs in machine language are run by the laws of physics (electrical engineering).

Compiling functions $C \colon L_1 \to L_2$ are notably useful if the translated program $C(p^{L_1})$ in L_2 in turn is the result of an executable program. Translating is a computational task and in principle determining $C(p^L)$ can be done by hand. But since many programs, also in a higher order programming language, may consist of several

[6]For example one may have a long series of translations:

$$L \to L_1 \to L_2 \to \cdots \to L_n \to M.$$

[7]Every student of a foreign language has to master this also: a stream of sounds

'papafumeunepipe'

has to be separated into words as follows 'papa fume une pipe'; only then one can translate further, into 'father smokes a pipe'.

million instructions, the computational task of compiling is better performed by a machine. A program that performs this translation is called a *compiler.* If such an automated translation process is of any use, the compiler needs to be written either in machine language M, or in another language L such that there is already an earlier compiler from L to M.

1.6. DEFINITION. Let $C^{L_1}\colon L_1 \to M$ be a compiling function. A *compiler* for C^{L_1} *written in language* L_2 is a program c^{L_1,L_2} such that

$$[\![c^{L_1,L_2}]\!]_{L_2} = C^{L_1}.$$

This is useful only if programs in L_2 are also executable. This is the case if $L_2 = M$ or if there is already a compiler from L_2 into M. Two cases will be important in this paper. 1. $L_2 = M$ and 2. $L_2 = L_1$.

1.3. Compilers written in machine language M

First we consider a compiler $c^L\colon L \to M$ written in machine language M.

1.7. PROPOSITION. *Let* $c^L\colon L \to M$ *be a compiler for a compiling function* C.

(i) *For all programs* p^L *written in* M *one has* $\{c^L\}_M(p^L) \twoheadrightarrow C(p^L)$.

(ii) *A computational job* $[\![p^L]\!]_L(x)$ *can be fully automated as follows.*

$$\{\{c^L\}(p^L)\}(x) \twoheadrightarrow \{C(p^L)\}(x) \twoheadrightarrow [\![p^L]\!]_L(x).$$

PROOF. (i) By Definition 1.6 we have $[\![c^L]\!]_M = C$. Hence by Proposition 1.2

$$\{c^L\}(p^L) \twoheadrightarrow [\![c^L]\!]_M(p^L) = C(p^L).$$

(ii) It follows that

$$\begin{array}{rcll} \{\{c^L\}(p^L)\}(x) & \twoheadrightarrow & \{C(p^L)\}(x), & \text{by (i)},\\ & \twoheadrightarrow & [\![p^L]\!](x), & \text{by Proposition 1.5.} \blacksquare \end{array}$$

1.8. DEFINITION. Let $c^L\colon L \to M$ be a compiler written in M.

(i) By Proposition 1.7(ii) there are two computation phases towards $[\![p^L]\!](x)$:

$$\{\{c^L\}(p^L)\}(x) \twoheadrightarrow^1 \{C(p^L)\}(x) \twoheadrightarrow^2 [\![p^L]\!](x).$$

The first computation 1, that is $\{c^L\}(p^L) \twoheadrightarrow C(p^L)$, takes place in a time interval that is called *compile-time*; the second computation 2, that is $\{C(p^L)\}(x) \twoheadrightarrow [\![p^L]\!](x)$, takes place in a time-interval that is called *run-time.*

(ii) If for programs p^L and inputs x (that interest us) the run-time $\{C(p^L)\}(x) \twoheadrightarrow [\![p^L]\!](x)$ is short (for our purposes), then the compiler c^L is said to *produce efficient code.* Note that this pragmatic definition depends only on the compiling function $C = [\![c^L]\!]$, and not on its program, the compiler itself.

(iii) If for programs p^L (that interest us) the compile-time is short (for our purposes), then the compiler is said to be *fast.* Note that this notion does depend on the compiler c^L, and not on the compiling function $C = [\![c^L]\!]$.

1.9. PROPOSITION. *For a programming language L, in which every program p^L is a sequence of statements consisting of a computable step, there exists a simple compiler $c_I^{L,M}\colon L \to M$ written in M for a compiling function C_I^L, mimicking the steps in L as steps in M. Such a compiler is called a (simple)* interpreter.

PROOF (Sketch). Let $p^L = s_1; s_2; \ldots; s_n$. Define $C_I^L(p^L) = I(s_1); I(s_2); \ldots; I(s_n)$, where $I(s)$ mimics the statement s by a (small) program in M. ■

For complex computational problems using a large program both the compile-time and run-time consume considerable amounts of time. Often these are bottlenecks for the feasibility of executing a program. Usually interpreters produce less efficient code than compilers to be discussed next.

1.4. Compilers written in higher programming languages

Now we consider the task of writing a compiler $c = c^{L,M}\colon L \to M$. If a compiler more complex than a simple interpreter is able to look at the input program p^L *in its totality* and can 'reflect' (act) on it, making optimizations for the run-time of the resulting code p^M. The goal is that such a compiler improves efficiency[8], using the power and flexibility of L. With the right effort a compiler can be developed that produces efficient code, so that to use such a compiler the run-time performance of the translated programs are optimized. This doesn't apply to the compile-time of compiler c is written in M, for which it is hard to achieve optimizations.

In his PhD thesis (1951) of just 50 pages Corrado Böhm designed a programming language L and constructed a compiler $c_B = c_B^{L,L}$, *in L itself.* This later made *bootstrapping* possible: producing not only efficient programs, but also making the compilation process itself efficient. We will explain how this is achieved. Suppose one has a compiler $c_B^{L,L} \in L$ that produces efficient code (efficiently running programs). Here 'efficient' is used in a non-technical intuitive sense. In order to run $c_B^{L,L}$ one needs a simple interpreter $c_I^{L,M}\colon L \to M$, written in M. Now we will describe three ways of computing the job $[\![p^L]\!]_L(x)$, that is, finding the result of an intended computation with program p^L written in L on input x.

1. Computing $[\![p^L]\!]_L(x)$ using the simple interpreter $c_I^{L,M}$:

$$\begin{array}{rcll} \{\{c_I^{L,M}\}(p^L)\}(x) & \twoheadrightarrow & \{C_I^L(p^L)\}(x), & \text{by 1 of Definition 1.8(i)},\\ & \twoheadrightarrow & [\![p^L]\!]_L(x), & \text{by 2 of Definition 1.8(i)}. \end{array}$$

This has both inefficient compile-time and run-time.

2. Better efficiency using $c_B^{L,L}$, run by the interpreter. Define $c_B^{L,M} = C_I^L(c_B^{L,L})$, the interpreter applied to the compiler written in L. This can be precompiled

$$c_B^{L,M} = C_I^L(c_B^{L,L}) \twoheadleftarrow \{c_I^{L,M}\}(c_B^{L,L}),$$

as the code of C_B^L in the sense that $[\![c_B^{L,M}]\!]_M = C_B^L$. One now has

$$\begin{array}{rcll} \{\{c_B^{L,M}\}(p^L)\}(x) & = & \{\{C_I^L(c_B^{L,L})\}(p^L)\}(x), & \text{by definition},\\ & \twoheadrightarrow & \{[\![c_B^{L,L}]\!]_L(p^L)\}(x), & \text{Prop. 1.5 applied to } C_I(c_B^{L,L}),\\ & = & \{C_B^L(p^L)\}(x), & \text{as } [\![c_B^{L,L}]\!]_L = C_B^L \text{ by definition},\\ & \twoheadrightarrow & [\![p^L]\!]_L(x), & \text{Prop. 1.5 applied to } C_B^L(p^L). \end{array}$$

[8]Software engineering studies ways to develop new versions of programs and compilers, in order to improve time performance and also to correct bugs (errors).

Computing $c_B^{L,M}$ is a one time job and, as the result can be stored, it doesn't count in measuring efficiency. The first computation $\twoheadrightarrow$ counts as the compile time of $c_B^{L,M}$. But it is also the run-time of $c_I^{L,M}$ (with compiling function C_I^L) and doesn't need to be efficient. The second computation $\twoheadrightarrow$ is the run time of c_B^L (with compiling function C_B^L) and was assumed to be efficient. Therefore this computation has an efficient run-time, but not necessarily an efficient compile-time.

3. Best efficiency using $c_B^{L,L}$: define $c_{B'}^{L,M} = C_B^L(c_B^{L,L})$, the compiler applied to itself. This can be precompiled as follows.

$$c_{B'}^{L,M} = C_B^L(c_B^{L,L}) \twoheadleftarrow \{c_B^{L,M}\}(c_B^{L,L}) \twoheadleftarrow \{\{c_I^{L,M}\}(c_B^{L,L})\}(c_B^{L,L}),$$

just requires a one time computation. Then again $[\![c_{B'}^{L,M}]\!]_M = C_B^L$, but now

$$\begin{array}{rcll} \{\{c_{B'}^{L,M}\}(p^L)\}(x) & = & \{\{C_B^L(c_B^{L,L})\}(p^L)\}(x), & \text{by definition,} \\ & \twoheadrightarrow & \{[\![c_B^{L,L}]\!]_L(p^L)\}(x), & \text{Prop. 1.5 applied to } C_B^L(c_B^{L,L}), \\ & = & \{C_B^L(p^L)\}(x), & \text{as } C_B^L = [\![c_B^{L,L}]\!]_L \text{ by definition,} \\ & \twoheadrightarrow & [\![p^L]\!](x), & \text{Prop. 1.5 applied to } C_B^L(p^L), \end{array}$$

with both efficient compile and run-time, as both codes have been generated by C_B^L.

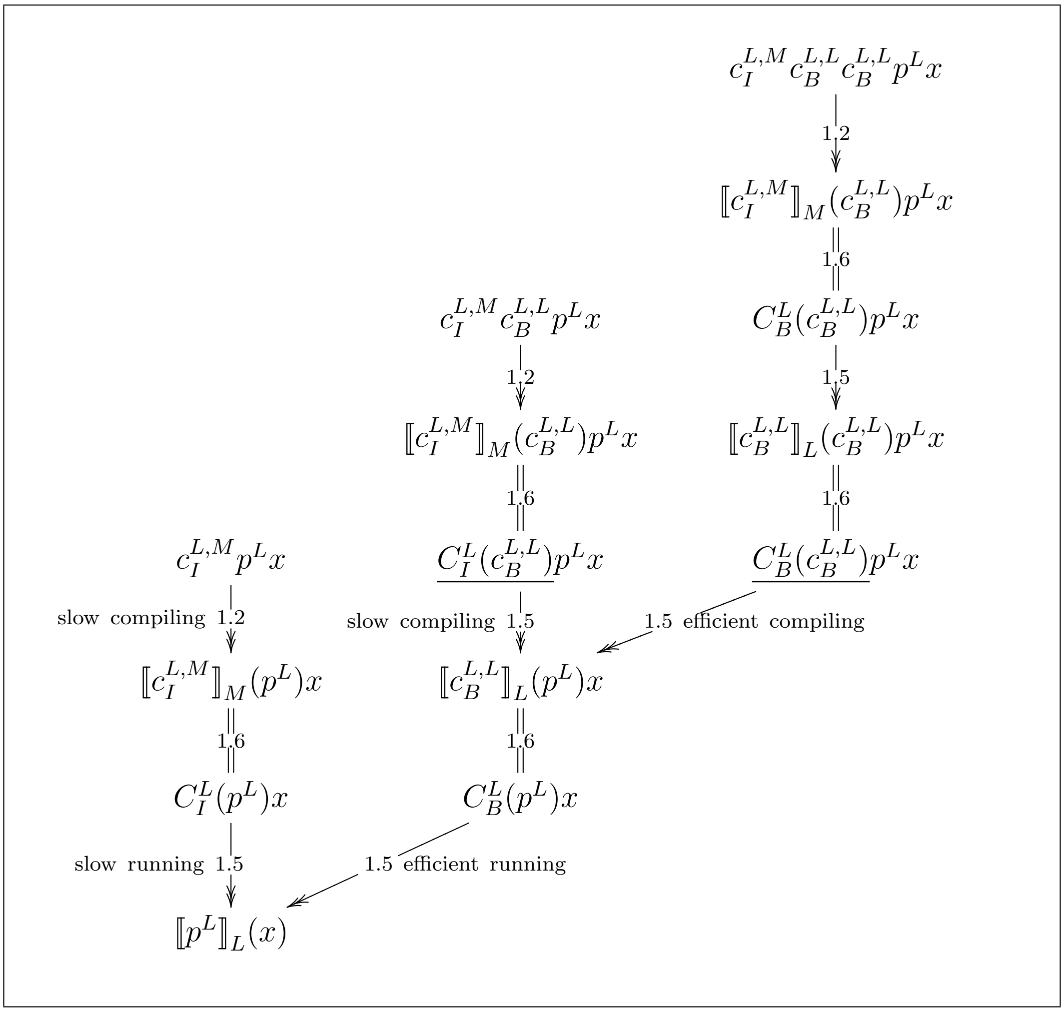
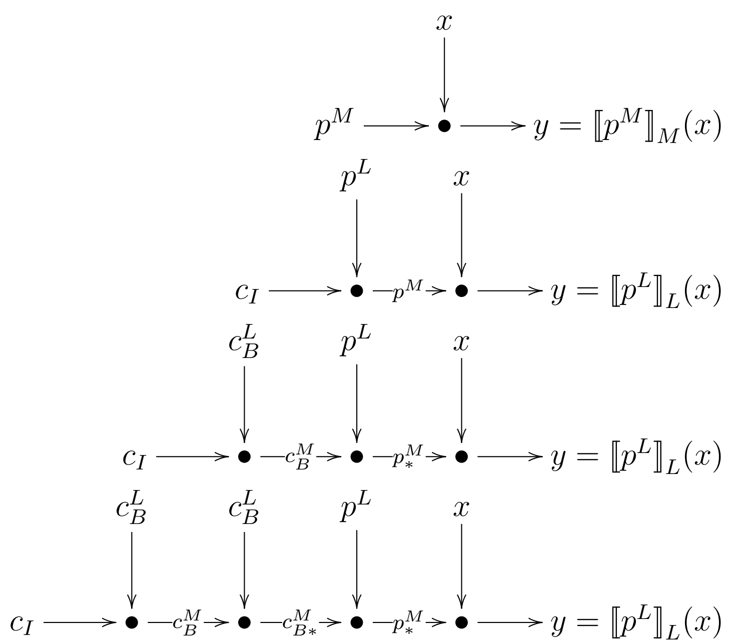


Figure 1: Bootstrapping: precompiled $c_B^{L,M} := C_I^L(c_B^{L,L})$, $c_{B'}^{L,M} := C_B^L(c_B^{L,L})$ provide efficient run time alone, or both run time and compile time, respectively.

In the language of combinatory logic, so much admired by Corrado Böhm, one writes $p \cdot x$, or simply px, for $\{p\}_M(x)$, and cpx for $(cp)x$, etcetera (association to the left). Then the three ways of compiling and computing a job $[\![p^L]\!]_L(x)$ can be rendered as in Figure 1. The underlined expressions denote the codes of the Böhm compiler $c_B^{L,L}$ that are obtained by precompilation, respectively using the interpreter and using itself. So the steps above these do not require time. This bootstrapping process wasn't in Böhm's PhD thesis, but it was made possible by his invention and realization of self-compilation.



The universal machine $\bullet$ with program p^M and input x. Although $M = L_M$ is a universal language, it is difficult to write creative programs in it.

Using a simple compiler $c_I\colon L \to M$ one can write better programs in L to run on M. These have slow compile-time and slow run-time.

Using c_B^L, with compiling function yielding optimized code, one can obtain efiicient run-time, but not compile-time.

Compiling c_B^L by itself yields an optimized c_{B*}^M from which one can obtain efiicient run-time and compile-time.

Figure 2: A different perspective on the same bootstrap process.

1.5. Compiler configurations

In this section we treat compilers in full generality translating a language L_1 into L_2. Only one machine M is used for the translation, but this easily can be generalized.

1.10. Definition. (i) We define the language $\mathscr{C}$ of *compiler configurations* by the following context free grammar.

$$\mathscr{C} ::= L \mid (L, \mathcal{C}_1, c, \mathcal{C}_2), \text{ where } c \text{ is a program in } L \text{ and } \mathcal{C}_1, \mathcal{C}_2 \in \mathscr{C}.$$

Actually L is a symbol for a language and c belongs to that language.

(ii) Let $\mathcal{C} \in \mathscr{C}$. The *language* of $\mathcal{C}$, in notation $|\mathcal{C}|$, is defined as follows.

$$\begin{aligned} |L| &= L; \\ |(L, \mathcal{C}_1, c, \mathcal{C}_2)| &= L. \end{aligned}$$

(iii) Correctness of $\mathcal{C} \in \mathscr{C}$ is defined as follows.

L is correct;
$(L, \mathcal{C}_1, c, \mathcal{C}_2)$ is correct if c is a program in programming language $|\mathcal{C}_2|$, $\mathcal{C}_1$, $\mathcal{C}_2$ are correct and $[\![c]\!]_{|\mathcal{C}_2|}\colon L \to |\mathcal{C}_1|$ is a compiling function.

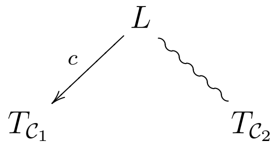
1.11. Example. The three situations in Subsection 1.4 can be described as compiler configurations. We use c_0 and c_B instead of $c_I^{L,M}$ and $c_B^{L,L}$, respectively.

$$
\begin{aligned}
\mathcal{C}_1 &= (L, M, c_0, M).\\
\mathcal{C}_2 &= (L, M, c_B, \mathcal{C}_1) = (L, M, c_B, (L, M, c_0, M)).\\
\mathcal{C}_3 &= (L, M, c_B, \mathcal{C}_2) = (L, M, c_B, (L, M, c_B, (L, M, c_0, M))).
\end{aligned}
$$

1.12. Definition. A compiler configuration $\mathcal{C}$ can be drawn as a labeled tree $T_{\mathcal{C}}$.

$$T_L = L;$$

$T_{(L,\mathcal{C}_1,c,\mathcal{C}_2)} =$



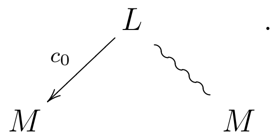
Compiler configurations and their trees are more convenient to use than the more rigid T-diagrams introduced in [43], since there is more flexibility to draw languages that still need to be translated. For example, $\mathcal{C}_3$ is the compiler configuration employed by Böhm and its tree explains well the magic trick.

1.13. Definition. A compiler configuration $\mathcal{C}$ is inductively defined to be *executable* as follows.

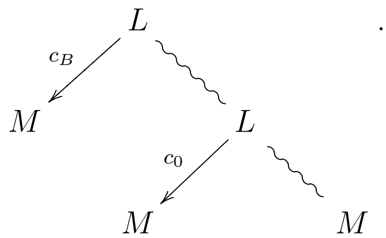
L	is executable iff $L = M$;
$(L, \mathcal{C}_1, c, \mathcal{C}_2)$	is executable iff $\mathcal{C}_1$ and $\mathcal{C}_2$ are executable.

1.14. Example. (i) The three compiler configurations $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ considered before are executable.

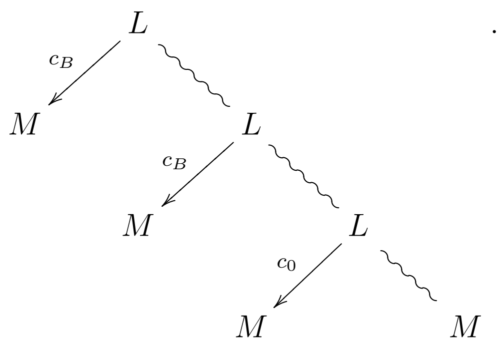
$T_{\mathcal{C}_1} =$



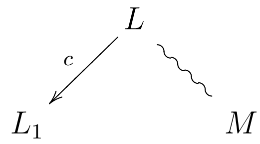
$T_{\mathcal{C}_2} =$

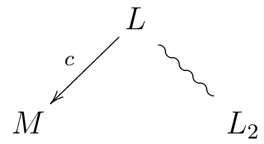


$T_{\mathcal{C}_3} =$



(ii) The following compiler configurations, drawn as trees, are not executable:

$T_{\mathcal{C}} =$ 

, $T_{\mathcal{C}'} =$ 

,

because no evaluation function for L_1 nor L_2 is given.

1.15. Definition. To each $\mathcal{C} \in \mathscr{C}$ we assign a function that maps a program p and value x to a value $\Phi_{\mathcal{C}}(p)(x)$, also written $\Phi_{\mathcal{C}} px$.

$$\begin{aligned} \Phi_L px &= [\![p]\!]_L(x); \\ \Phi_{L,\mathcal{C}_1,c,\mathcal{C}_2} px &= \Phi_{\mathcal{C}_1}(\Phi_{\mathcal{C}_2} cp)x. \end{aligned}$$

1.16. Exercise. For all correct and executable $\mathcal{C} \in \mathscr{C}$, $p \in |\mathcal{C}|$, $x \in D$ one has

$$\Phi_{\mathcal{C}} px = [\![p]\!]_{|\mathcal{C}|}(x).$$

1.17. Example. In the following evaluations we leave out parenthesis, like in lambda calculus and combinatory logic.

$$\begin{array}{rclll} \Phi_M p^M x &=& [\![p^M]\!]_M x & \twoheadleftarrow \{p^M\}x & = p^M x. \\ \Phi_{\mathcal{C}_1} p^L x &=& [\![[\![c_I]\!]_M p^L]\!]_M x & \twoheadleftarrow \{\{c_I\}p^L\}x & = c_I p^L x. \\ \Phi_{\mathcal{C}_2} p^L x &=& [\![[\![[\![c_I]\!]_M c_B]\!]_M p^L]\!]_M x & \twoheadleftarrow \{\{\{c_I\}c_B\}p^L\}x & = c_I c_B p^L x. \\ \Phi_{\mathcal{C}_3} p^L x &=& [\![[\![[\![[\![c_I]\!]_M c_B]\!]_M c_B]\!]_M p^L]\!]_M x & \twoheadleftarrow \{\{\{\{c_I\}c_B\}c_B\}p^L\}x & = c_I c_B c_B p^L x. \end{array}$$

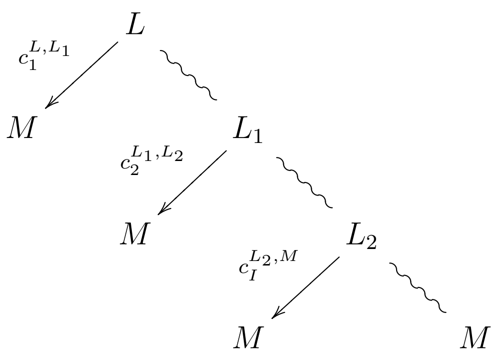
Do we absolutely need self-compilation in order to obtain efficient compilation? The answer is negative. Suppose one has the following:

1. a compiler $c_1^{L,L_1} : L \to M$, producing fast code, written in L_1;
2. a compiler $c_2^{L_1,L_2} : L_1 \to M$, producing fast code written in L_2;
3. a simple interpreter $c_I^{L_2,M} : L_2 \to M$, written in M.

Then one can form the following correct and executable compiler configuration:

$$\mathcal{C}_4 = (L, M, c_1^{L,L_1}, (L_1, L_2, c_2^{L_1,L_2}, (L_2, M, c_I^{L_2,M}, M))),$$

with tree

$T_{\mathcal{C}_4} =$ 

.

Again one obtains a compiler with fast compile-time that produces efficient code

$$c = C_2(c_1^{L,L_1}) \twoheadleftarrow \{\{c_I^{L_2,M}\}(c_2^{L_1,L_2})\}(c_1^{L,L_1}).$$

In the magic trick of Böhm, compiler (3) in Subsection 1.4 above, he took $L = L_1 = L_2$ and $c_1^{L,L_1} = c_2^{L_1,L_2} = c_B^{L,L}$. This saves work: only one language and one compiler need to be developed.

After having obtained his PhD in Zürich, Böhm did obtain a patent on compilers. But, unexpectedly, a few years later (1955) IBM came with its FORTRAN compiler. It turned out that Böhm's patent was valid only in Switzerland!

2. Structured programming

In a Turing machine transition a state can be followed by any other state. Therefore many programming languages naturally contain the '`goto`' statement. When these are used in a mindless way, the meaning and hence correctness of programs is much more difficult to warrant. The first half of the paper of Böhm and Jacopini [17] is dedicated to eliminate goto statements, as a first step towards structured programs. That part of the paper is stated to be written by Jacopini, but I think we may suppose that Böhm, the supervisor of Jacopini, has contributed to it.

2.1. Imperative programming

The Universal Turing Machine, or an improved version, immediately gives rise to a language with `goto` statements: the machine, being in state s_1 changes (under the right conditions) into state s_2. This is expressed by a statement very much like a 5-tuple of a Turing Machine $\langle \mathtt{1}, \mathtt{s}_1, \mathtt{0}, \mathtt{s}_2, \mathtt{N} \rangle$, that in the presence of named registers looks like

$$\mathtt{s}_1\text{: } \mathtt{if}\ \mathtt{x} = \mathtt{1}\ \mathtt{then}\ \mathtt{x} := \mathtt{0};\ \mathtt{goto}\ \mathtt{s}_2;$$

Here the meaning is as follows: the machine checks whether the content of register x equals 1 and then it overwrites the 1 by a 0 as the content of register x, after which it jumps to state $\mathtt{s}_2$. In the presence of addressable registers like $\mathtt{x}$, there is no longer a need to use the small step local movements indicated by $\{\mathtt{L}, \mathtt{R}, \mathtt{N}\}$. A more extended example is the following.

$$\mathtt{s}_1\text{: } \mathtt{if}\ \mathtt{x} = \mathtt{1}\ \mathtt{then}\ (\mathtt{y} := \mathtt{0};\ \mathtt{goto}\ \mathtt{s}_2)\ \mathtt{else}\ (\mathtt{y} := \mathtt{y} + \mathtt{1};\ \mathtt{goto}\ \mathtt{s}_3);$$

Apart from branching, leading naturally to a flow-chart as a representation of such a program, we also see the for imperative programming typical statement $\mathtt{y} := \mathtt{y} + \mathtt{1}$, meaning that the content of register y is overwritten by the old content augmented by one. Many such components can form nice-looking but hard to understand diagrams. One can imagine that the idea arose to create more understandable diagrams and as a first step to eliminate the `goto` statements.

2.2. Eliminating the '`goto`'

In this subsection it is shown that the result of eliminating the `go to` statement can be seen in the light of Kleene's analysis of computability, as was pointed out by Harel [32], but also by Cooper [24].

2.1. Theorem (Kleene Normal Form Theorem). *There are functions U, T that are primitive computable such that every computable function f has a code number e such that for all $\vec{x} \in \mathbb{N}$ one has*

$$f(\vec{x}) = U(\mu z.T(e, \vec{x}, z) = 0). \qquad \text{(NFT)}$$

If P is a predicate on $\mathbb{N}$, then $\mu z.P(z)$ denotes the least number $z \in \mathbb{N}$ such that $P(z)$, if this z exists, otherwise the expression is undefined. In (NFT) it is assumed that for all x there exists a z such that $T(e,x,z)$ holds[9].

PROOF (Sketch). The value of the function $f(\vec{x}) = y$ can be computed by the Universal Turing Machine $\mathcal{U}$ using, say, e as program. Then there is a computation

$$(\mathtt{input}, \mathtt{s}_0, \mathtt{p}_0) \to_{\mathcal{U}} (\mathtt{t}_1, \mathtt{s}_1, \mathtt{p}_1) \to_{\mathcal{U}} \cdots \to_{\mathcal{U}} (\mathtt{t}_\mathtt{k}, \mathtt{s}_\mathtt{k}, \mathtt{p}_\mathtt{k}) \to_{\mathcal{U}} (\mathtt{output}, \mathtt{s}_\mathtt{h}, \mathtt{p}_\mathtt{h}), \quad \text{(comp)}$$

where $\mathtt{input} = (e, \vec{x})$, '$\mathtt{input}, \mathtt{s}_0, \mathtt{p}_0$' is the first configuration, '$\mathtt{output}, \mathtt{s}_0, \mathtt{p}_0$' is the last one that is terminating, and $\mathtt{output} = y$. Furthermore, T is the characteristic function ($= 0$ when true, $= 1$ when false) of the primitive computable predicate $P(e, \vec{x}, z)$, that holds if z is (the code of) the computation (comp). After a search (by μ) for this (coded sequence) z, the $y = \mathtt{output}$ is easily obtainable from it, which is done by the primitive computable function U. ■

2.2. THEOREM (Böhm-Jacopini [17]). *A program built up from statements of the form*

```
x:=x+1
x:=x-1
if B, then S1 else S2
goto q
```
$\Big\}\ L_1$

can be replaced by an equivalent one built up from statements of the form

```
x:=x+1
x:=x-1
if B, then S1 else S2
for k:=0 to n do A(k)
while x>0 do A(x)
```
$\Big\}\ L_2$

PROOF (Sketch). A function f with program from L_1 will be computable by the universal Turing Machine by program, say, e. Therefore by Theorem 2.1 one has $f(\vec{x}) = U(\mu z.T(e, \vec{x}, z) = 0)$. The functions U, T are primitive computable, hence expressible by the '`for`' statements. Only for the μ the `while` statements are needed. (Actually this happens only a single time.) ■

2.3. COROLLARY (Folk Theorem). *Programs in L_1 can be replaced by an equivalent one in L_2 using the* `while` *construct only a single time.*

PROOF. By the parenthetical remark in the proof of 2.2. ■

2.3. Evaluation

After the `goto` was shown to be eliminable, in Dijkstra's note [30] a polemics was started '`goto` statement considered harmful'. In the book [29] structured programming was turned into an art. Knuth [38] argued that eliminating the `goto` as in the above proof of Theorem 2.2 may produce unstructured programs, unrelated to the original program. The original proof in [17] preserves the structure of the program.

[9] The formula (NFT) also holds for partial functions f, in which case $f(\vec{x})\uparrow$ iff $\forall z.T(e, \vec{x}, z) \neq 0$.

See [44] for a detailed exposition of this paper. An even better way to eliminate the `goto` statements, while preserving the structure of a program, is Ashcroft and Manna [1]. Knuth [38] also gives an example of a program in which a `goto` statement improves its structure.

In Harel [32] the paper of Böhm and Jacopini [17] was taken as an example of how a 'Folk Theorem' appears. The result attributed to these authors often is Corollary 2.3, rather than Theorem 2.2 itself.

As remarked in [17] it seems necessary to use an extra variable to obtain a program without a `goto`, but the authors couldn't find a proof of this conjecture. It was proved by Ashcroft and Manna [1], but also in Knuth and Floyd [39] and Kozen and Tseng [40].

Although the Böhm-Jacopini result started a discussion towards structured programming, a new idea was needed to obtain even better structured programs. As we will see in the next section, actually it was an old idea: functional programming based on lambda calculus.

3. Functional programming and the CUCH machine

It was Wolf Gross, colleague of Corrado Böhm, who introduced the latter to functional programming based on type-free lambda calculus, in which unbounded self-application is possible. As can be imagined, knowing the construction of a self-applicative compiler, it had a deep impact on the sequel of Böhm's professional life. As there is a paper Intrigila-Mazzucchelli in this memorial volume on Böhm's contribution to functional programming, we restrict ourselves to give some historic and conceptual background.

3.1. Functional programming

Alonzo Church introduced lambda calculus as a way to mathematically characterize the intuitive notion of computability. I seem to remember that he told me the following story. Church's thesis supervisor, Oswald Veblen, gave him the problem to compute the Betti numbers of an algebraic surface given by a polynomial equation. Church did not succeed in this task and was stuck developing his PhD thesis. He then did what other mathematicians do in similar circumstances: solve a different but related problem. Church wondered what the notion 'computable' actually means. Perhaps determining the Betti number of a surface from its description is not computable.

Church then introduced a formal system for mathematical deduction and computation [18, 19]. In [36] his students Kleene and Rosser found an inconsistency[10] in Church's original system. After that Church [20] stripped the system from the deductive part and obtained the (pure) lambda calculus, which turned out to be provably consistent [22]. See [3] for an extensive exposition of the lambda calculus.

To formally define the notion of computability, Church introduced numerals $\mathbf{c}_n$ representing natural numbers n as lambda terms. Rosser found ways to add, multiply and exponentiate: that is, he found terms $A_+, A_\times, A_{\mathsf{exp}}$ such that $A_+\mathbf{c}_n\mathbf{c}_m \twoheadrightarrow \mathbf{c}_{n+m}$, and similarly for multiplication and exponentiation. This way these three functions were seen to be lambda definable. Here $\twoheadrightarrow$ denotes many-step rewriting,

[10] The proof of a contradiction in Church's system was essentially simplified by Curry [28].

the transitive reflexive closure of $\to$. At first Church nor his students could find a way to show that the predecessor function was lambda definable. At the dentist's office Kleene did see how to simulate recursion by iteration and could in that way construct a term lambda defining the predecessor function, [26]. (I believe Kleene told me it was under the influence of laughing gas, NO, used as anesthetic.) When Church saw that result he stated "Then all intuitively computable functions must be lambda definable." That was the first formulation of Church's thesis and the functional model of computation was born. At the same time Church gave an example of a function that was non-computable in this model.

Turing [53] proved that the imperative and functional models of computation have the same power: they can compute exactly the same partial functions, on say the natural numbers. The way these computations are performed, however, differs considerably. In both cases computations traverse a sequence of configurations, starting essentially from the input leading to the output. But here the common ground ends.

3.2. Comparing imperative and functional programming

In functional programming the argument(s) A (or $\vec{A}$) for a computation in the form of a function F that has to be applied to them form one single expression FA (respectively $F\vec{A}$). Such expressions are subject to rewriting. If the expression cannot be rewritten any further, then the so called *normal form* has been reached and this is the intended output. The intermediate results all have the same meaning as the original expression and as the output. A basic example of this is

$$(\lambda x.x^2 + 1)(3) \to 3^2 + 1 \to 9 + 1 \to 10, \tag{1}$$

where $(\lambda x.x^2{+}2)$ is the function $x \mapsto x^2{+}1$ that assigns to x the value $x^2{+}1$. In more complex expressions there is a choice of how to rewrite, that is, which subexpression to choose as focus of attention for elementary steps as above. For example not all choices will lead to a normal form. There are *reduction strategies* that always will find a normal form if it exists. Normal forms, if they are reached, are unique, the result is independent of choices how to rewrite. However performance, both time and space, is sensitive to the steps employed.

In the imperative model a computation the configurations at each moment of a computation sequence of a Turing Machine M consist of the momentaneous memory content on the *tape*, the *state* of M, and *position* of its head: $(\mathtt{t}, \mathtt{s}, \mathtt{p})$. Each terminating computation runs as follows:

$$(\mathtt{input}, \mathtt{s_0}, \mathtt{p_0}) \to_{\mathtt{M}} (\mathtt{t_1}, \mathtt{s_1}, \mathtt{p_1}) \to_{\mathtt{M}} \cdots \to_{\mathtt{M}} (\mathtt{t_k}, \mathtt{s_k}, \mathtt{p_k}) \to_{\mathtt{M}} (\mathtt{output}, \mathtt{s_h}, \mathtt{p_h}), \tag{IP}$$

where $\mathtt{s_h}$ is a halting state (and $\mathtt{p_h}$ is irrelevant). The transitions $\to_{\mathtt{M}}$ depend on the set of instructions of the Turing Machine M. In the case of non-termination the configurations never reach one with a terminal state. This description already shows that, wanting to combine Turing Machines to form one that is performing a more complex task, requires some choices of e.g. making the final state of the first machine fit with the initial one of the second machine.

In the functional model of computation the sequence of configurations is as follows:

$$\mathtt{F\,input} \to_\beta \mathtt{E_1} \to_\beta \cdots \to_\beta \mathtt{E_k} \to_\beta \mathtt{output}. \tag{FP}$$

All of these configurations are λ-terms and the transitions $\to_\beta$ are according to the single β-rule of reduction, which is quite different. In order to make a more fair comparison between the imperative and functional computation, one could change (IP) and denote it as

$$(\mathtt{input}, \mathtt{c}, \mathtt{s}_0, \mathtt{p}_0) \to_{\mathcal{U}} (\mathtt{t}_1, \mathtt{c}, \mathtt{s}_1, \mathtt{p}_1) \to_{\mathcal{U}} \cdots \to_{\mathcal{U}} (\mathtt{t}_\mathtt{k}, \mathtt{c}, \mathtt{s}_\mathtt{k}, \mathtt{p}_\mathtt{k}) \to_{\mathcal{U}} (\mathtt{output}, \mathtt{c}, \mathtt{s}_\mathtt{h}, \mathtt{p}_\mathtt{h}) \qquad (\mathrm{IP}'),$$

where c is the code (program) that makes the universal machine $\mathcal{U}$ imitate the machine M. This makes (IP′) superficially similar to (FP).

Advantages of functional programming

But there are essential differences between the two models of computation. First of all, in the sequence (FP) the expressions are words in a language more complex than the simple strings in (IP) or (IP′).

(i) The λ-terms expressing functional programs have the possibility of making abstraction upon abstraction arbitrarily often. This means that 'components' of functions can be also functions (of functions), enabling flexible procedures.

(ii) In FP there is no mention of state and position, hence there is no need to deal with the bureaucracy of these when combining programs. Hence FP has easy compositionality.

(iii) In the sequence (FP) the meaning of each configuration remains the same, from the first to the last expression. This can be seen clearly in the sequence (1) above.

Features (i) and (ii) of functional programs makes them transparent and compact. Feature (iii) makes it easier to prove them correct: reasoning with mathematical induction, substitution and abstraction often suffice; no need to learn new logical formalisms to analyze imperative programs. It can be expected that FP will become more and more important. The lack of side-effects makes it more easy to make parallel versions of programs.

Implementations of functional programming

Functional Programming has been developed much more slowly than Imperative Programming. The reason is that imperative programs can be implemented rather directly on a Turing Machine or modern computer. This is not the case for functional programs. Attempts to develop specialized hardware for Functional Programming have not been successful. But over the years compilers from functional languages into ordinary CPU's for imperative programs have been developed.

One of the first examples is the SECD machine of Landin [41], soon followed by the work of Böhm and Gross on the CUCH machine, [16], [12]. After fifty years of research on the use and implementation of functional programming the field has come of age. There exist fast compilers producing efficient code. One can focus on the mathematical definition of the functions involved and the correctness of these can be proved with relatively simple tools, like substitution, abstraction and induction. A functional program is automatically structured. There are for example no '`goto`' statements. See [9] for a short description, [34] for an extensive motivation, and [48] for implementing functional programming languages.

Challenges for functional programming

There are two challenges for functional programming languages. 1. The lack of state makes writing code for input/output more complex. The best known functional languages are LISP, [42] (with many modern versions starting with SCHEME [51], and ML [45] (with modern version OCaml [47]). ML is loosely characterized as 'LISP with types' coming of the simply types lambda calculus, see [21, 27], and in modern form [8], Part I. LISP and ML are not pure functional programming languages, in that they have assignment statements that can be used for input and output. But this makes it also possible to write unstructured programs. In the pure functional languages, Haskell [33] and Clean [23], at present the most developed ones, the I/O problem is solved by respectively monads and uniqueness typing. But using these features, in both cases it is still possible to write incomprehensible code when dealing with I/O. 2. The evaluation result, the output, doesn't depend on the way reduction takes place, but it is not always easy to reason about space and time efficiency. These issues are beyond the scope of this paper.

4. Separability in λ-calculus

A mathematician is interested in numbers, not because these may represent the amount of money in one's bank accounts, but for their properties definable using the basic arithmetical operations, such as primality. Although such a love for numbers is obvious for a number theorist, and almost offensive to mention, this is not the case for most people. In the same way Corrado Böhm became interested in λ-terms, not because they represent programs that one can sell, but for their properties definable from the basic operations in lambda calculus, usually only application. This is somewhat different from the love of say Donald Knuth for imperative programs, obvious from his volumes [37], that is driven by the challenge to write clear, elegant, and efficient algorithms that perform relevant computational tasks.

We assume elementary knowledge of lambda calculus and recall the following notations.

Notation. (i) The set of all lambda terms is denoted by Λ. The set of free variables of $M \in \Lambda$ is denoted by $\mathrm{FV}(M)$. The set of closed lambda terms is defined by $\Lambda^{\emptyset} = \{M \in \Lambda \mid \mathrm{FV}(M) = \emptyset\}$.

(ii) '$\equiv$' denotes equality up to renaming bound variables, e.g. $\lambda x.x \equiv \lambda y.y$.

(iii) '$=$' denotes β-convertibility on λ-terms, also denoted by '$=_\beta$' to be explicit.

(iv) $=_\beta$ is generated by β-reduction $\to_\beta$, as in $(\lambda x.M)N \to_\beta M[x := N]$.

(v) $=_\eta$ is generated by η-reduction $\to_\eta$, as in $\lambda x.Mx \to_\eta M$.

(vi) $M \in \Lambda$ is in $\beta(\eta)$ *normal form* ($\beta(\eta)$-nf) if no $\to_\beta$ (nor $\to_\eta$) step is possible.

(vii) For $M_1, \ldots, M_n \in \Lambda$ write $\langle M_1, \ldots, M_n\rangle = \lambda z.zM_1 \cdots M_n$, with z a fresh variable, i.e. $z \notin \mathrm{FV}(M_1 \cdots M_n)$.

(viii) Write $\mathsf{U}^n_k = \lambda x_1 \cdots x_n.x_k$. Note that $\langle M_1, \ldots, M_n\rangle \mathsf{U}^n_k = M_k$ for $1 \le k \le n$.

(ix) Write

$$\begin{array}{rcll} \mathsf{I} &=& \lambda x.x; & \\ \mathsf{K} &=& \lambda xy.x, & \text{serving as `}\texttt{true}\text{'}; \\ \mathsf{K}_* &=& \mathsf{KI} =_\beta \lambda xy.y, & \text{serving as `}\texttt{false}\text{'}; \\ \mathsf{S} &=& \lambda xyz.xz(yz); & \end{array}$$

$$
\begin{array}{rcll}
\mathsf{C} & = & \lambda xyz.xzy; & \\
\mathsf{Y} & = & \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)); & \text{Curry's fixed point combinator;} \\
\Theta & = & (\lambda ab.b(aab))(\lambda ab.b(aab)), & \text{Turing's fixed point combinator;} \\
\omega & = & \lambda x.xx; & \\
\Omega & = & \omega\omega, & \text{standard term without a nf;} \\
\mathbf{c}_k & = & \lambda fx.f^k x, & \text{where } f^0x = x \text{ and } f^{n+1}x = f(f^n x) \\
 & & & \text{(Church's numerals).}
\end{array}
$$

Separability of two normal forms

4.1. Definition. Terms $M_0, M_1 \in \Lambda^{\emptyset}$ are called *separable* if for all $P_0, P_1 \in \Lambda^{\emptyset}$ there exists an $F \in \Lambda^{\emptyset}$ such that

$$FM_0 =_\beta P_0 \ \&\ FM_1 =_\beta P_1.$$

This is equivalent to requiring that there is a lambda definable bijection

$$F\colon \{M_0, M_1\} \to \{\mathbf{c}_0, \mathbf{c}_1\}$$

with lambda definable inverse, in which case we write $\{M_0, M_1\} =_1 \mathbf{c}_0, \mathbf{c}_1$.

In result 4.3 the principal step was proved by Böhm [13] with the following result.

4.2. Theorem (Böhm [13]). *Let $M_0, M_1 \in \Lambda^{\emptyset}$ be two different λ-terms in $\beta\eta$-nf. Then for all $P_0, P_1 \in \Lambda^{\emptyset}$ there exist $\vec{N} \in \Lambda^{\emptyset}$ such that*

$$
\begin{array}{rcl}
M_0\vec{N} & = & P_0, \\
M_1\vec{N} & = & P_1.
\end{array}
$$

Proof (Sketch). A full proof can be found in [3, Theorem 10.4.2] and in [31] an intuitive one with applications. Idea: give the M, N arguments separating the two; as we do not know in advance which arguments will work, we may use variables and specify them later. We present some examples.

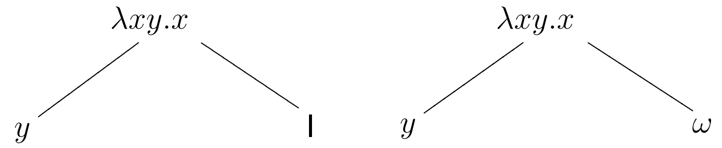
Example 1. I, K.

$$
\begin{array}{l|l|l|l|l}
 & \scriptstyle xy & \scriptstyle x:=\mathsf{KK} & \scriptstyle zvw & \scriptstyle z:=\mathsf{I} \\
\mathsf{I} & \mathsf{I}xy = xy & \mathsf{KK}y = \mathsf{K} & \mathsf{K}zvw = zw & w \\
\mathsf{K} & \mathsf{K}xy = x & \mathsf{KK} & \mathsf{KK}zvw = \mathsf{K}vw = v & v
\end{array}
\qquad
\text{Hence}\quad
\begin{array}{rcl}
\mathsf{I}(\mathsf{KK})\mathsf{I}vw & = & w; \\
\mathsf{K}(\mathsf{KK})\mathsf{I}vw & = & v.
\end{array}
$$

Example 2. I, ω.

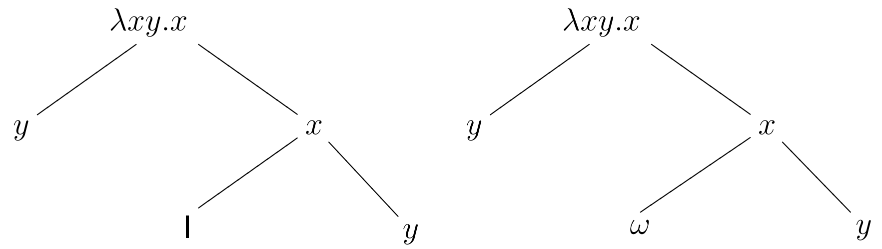
$$
\begin{array}{l|l|l|l|l}
 & \scriptstyle x & \scriptstyle x:=\mathsf{K}_* & \scriptstyle xyz & \scriptstyle x:=\mathsf{K}_*,\, y:=\mathsf{K}x \\
\mathsf{I} & \mathsf{I}x = x & \mathsf{K}_* & \mathsf{K}_*xyz = yz & x \\
\omega & \omega x = xx & \mathsf{K}_*\mathsf{K}_* = \mathsf{KIK}_* = \mathsf{I} & \mathsf{I}xyz = xyz & z
\end{array}
\qquad
\text{Hence}\quad
\begin{array}{r}
\mathsf{IK}_*\mathsf{K}_*(\mathsf{K}x) = x; \\
\omega\mathsf{K}_*\mathsf{K}_*(\mathsf{K}x) = z.
\end{array}
$$

Example 3. $M \triangleq \lambda xy.xy\mathsf{I}$, $N \triangleq \lambda xy.xy\omega$. Consider these as trees:

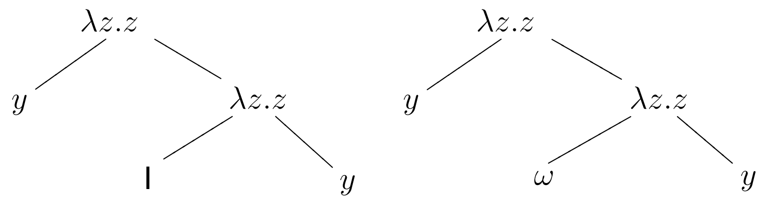


In order to separate these, we zoom in on the difference I and ω, via $M\mathsf{K}_*y, N\mathsf{K}_*y$, giving I, ω respectively, and we know how to separate these by Example 2.

Example 4. $M \triangleq \lambda xy.xy(x\mathsf{I}y)$, $N \triangleq \lambda xy.xy(x\omega y)$. Consider their trees:



Again we like to zoom in on the difference I and ω. Dilemma: one cannot make the x choose both left and right. Solution: applying the 'Böhm transformation' xy, $x{:=}\lambda abz.zab$ gives trees



and one can zoom in by application to $z, z{:=}\mathsf{K}_*, z, z{:=}\ \mathsf{K}$, obtaining I and ω and we are back to Example 2. Note that the dilemma was solved by first 'getting rid of x, y' and then 'substituting $\lambda z.z$ for x' enabling to make postponed choices: first K (going right), then K_* (going left). ∎

It is clear that one needs to require that the terms have different $\beta\eta$-nfs, not just β-nfs. The terms $\lambda x.x$ and $\lambda xy.xy$ are different β-nfs, but cannot be separated: $F(\lambda x.x) = \lambda xy.x, F(\lambda xy.xy) = \lambda xy.y$ would imply

$$\lambda xy.x =_\beta F(\lambda x.x) =_\eta F(\lambda xy.xy) =_\beta \lambda xy.y,$$

from which any equation can be derived, contradicting that the $\lambda\beta\eta$-calculus is consistent.

4.3. Corollary ([52]). *For all $M_0, M_1 \in \Lambda^\varnothing$ having a β-nf the following are equivalent.*

(i) *For all $P_0, P_1 \in \Lambda^\varnothing$ there exist $\vec{N} \in \Lambda^\varnothing$ such that*

$$M_0\vec{N} =_\beta P_0 \ \&\ M_1\vec{N} =_\beta P_1.$$

(ii) *M_0, M_1 are separable, i.e. for all $P_0, P_1 \in \Lambda^\varnothing$ there exists an $F \in \Lambda^\varnothing$ such that*

$$FM_0 =_\beta P_0 \ \&\ FM_1 =_\beta P_1.$$

(iii) *There exists an $F \in \Lambda^\varnothing$ such that*

$$FM_0 =_\beta \lambda xy.x \ \&\ FM_1 =_\beta \lambda xy.y.$$

(iv) *The equation $M_0 = M_1$ is inconsistent with $\lambda\beta$.*

(v) *The equation $M_0 = M_1$ is inconsistent with $\lambda\beta\eta$.*

(vi) *The terms M_0, M_1 have distinct $\beta\eta$-nfs.*

PROOF. (i)⇒(ii) By (i) there are $\vec{N}$ such that $M_i\vec{N} =_\beta P_i$. Take $F := \lambda m.m\vec{N}$.

(ii)⇒(iii) Take $P_i := \lambda x_0x_1.x_i$, for $0 \leq i \leq 1$.

(iii)⇒(iv) From the equation $M_0 = M_1$ one can by (iii) derive $\lambda xy.x = \lambda xy.y$, from which one can derive any equation; all derivations using just $\lambda\beta$.

(iv)⇒(v) Trivial.

(v)⇒(vi) By the assumption that M_0, M_1 have β-nfs and [3], Corollary 15.1.5, it follows that M_0, M_1 have $\beta\eta$-nfs. If these were equal, then $M_0 =_{\beta\eta} M_1$ and hence $M_0 = M_1$ would be consistent, contradicting (v).

(vi)⇒(i) By Theorem 4.2. ■

Separability of finite sets of normal forms

Together with his students Böhm generalized this in [15] from two to k terms.

4.4. DEFINITION. A finite set $\mathcal{A} \subseteq \Lambda^\varnothing$ is called *separable* if for some $k \in \mathbb{N}$

$$\mathcal{A} =_1 \{\mathbf{c}_0, \ldots, \mathbf{c}_{k-1}\}.$$

4.5. THEOREM ([15]). *Let $M_0, \ldots, M_{k-1} \in \Lambda^\varnothing$ be terms having different $\beta\eta$-nfs. Then $\{M_0, \ldots, M_{k-1}\}$ is separable. One even has for all terms $P_0, \ldots, P_{k-1} \in \Lambda^\varnothing$ there exist terms $\vec{N} \in \Lambda^\varnothing$ such that*

$$\begin{array}{rcl} M_0\vec{N} & =_\beta & P_0, \\ & \ldots & \\ M_{k-1}\vec{N} & =_\beta & P_{k-1}. \end{array}$$

PROOF. For a proof see [15] or [3, proof of Corollary 10.4.14.].

4.6. COROLLARY. *Let $\mathcal{A} \subseteq \Lambda^\varnothing$ be a finite set of terms all having a β-nf. Then*

$\mathcal{A}$ is separable $\Leftrightarrow$ the $\beta\eta$-nfs of the elements of $\mathcal{A}$ are mutually different.

Separability of finite sets of general terms

A characterization of separability for finite $\mathcal{A} \subseteq \Lambda^\varnothing$, possibly containing terms without normal form, is due to Coppo, Dezani, and Ronchi [25], see also [3], Theorem 10.4.13. To taste a flavor of that theorem we give some of its consequences coming from [52].

1. The set $\left\{ \begin{array}{l} \lambda x.x\mathbf{c}_0\Omega, \\ \lambda x.x\mathbf{c}_1\Omega \end{array} \right\}$ is separable; so is $\left\{ \begin{array}{l} \lambda xy.xx\Omega, \\ \lambda xy.xy\Omega \end{array} \right\}$.

2. $\left\{ \begin{array}{l} \lambda x.x(\lambda y.y\Omega), \\ \lambda x.x(\lambda y.y\mathbf{c}_0) \end{array} \right\}$ is not separable; neither is $\left\{ \begin{array}{l} \lambda x.x, \\ \lambda xy.xy \end{array} \right\}$.

3. $\left\{ \begin{array}{l} \lambda x.x(\lambda y.y\mathbf{c}_0\Omega(\lambda z.z\Omega)), \\ \lambda x.x(\lambda y.y\mathbf{c}_1\Omega(\lambda z.z\mathbf{c}_1)), \\ \lambda x.x(\lambda y.y\mathbf{c}_1\Omega(\lambda z.z\mathbf{c}_2)) \end{array} \right\}$ is separable.

4. $\left\{ \begin{array}{l} \lambda x.x\mathbf{c}_0\mathbf{c}_0\Omega, \\ \lambda x.x\mathbf{c}_1\Omega\mathbf{c}_1, \\ \lambda x.x\Omega\mathbf{c}_2\mathbf{c}_2 \end{array} \right\}$ is not separable, although each proper subset is.

Separability of infinite sets of general terms

In [52] for infinite sets separability is defined and characterized. Here we give a slightly alternative formulation.

4.7. NOTATION. Let $\mathcal{A} \subseteq \Lambda^{\emptyset}$. Write for $F \in \Lambda^{\emptyset}$

$$F\mathcal{A} \triangleq \{FM \mid M \in \mathcal{A}\};$$
$$\mathcal{C}_{\mathbb{N}} \triangleq \{\mathbf{c}_n \mid n \in \mathbb{N}\}.$$

4.8. DEFINITION. Let $\mathcal{A} \subseteq \Lambda^{\emptyset}$ be an infinite set. Then

(i) $\mathcal{A}$ is called *special* if there are combinators $F, G \in \Lambda^{\emptyset}$ such that modulo $=_\beta$ one has

$$\begin{array}{ll} F{:}\,\mathcal{A} \to \mathcal{C}_{\mathbb{N}} & \text{is an injection,} \\ G{:}\,\mathcal{C}_{\mathbb{N}} \to \mathcal{A} & \text{is a surjection.} \end{array}$$

(ii) $\mathcal{A}$ is called *separable* if $\mathcal{A} =_1 \mathcal{C}_{\mathbb{N}}$, that is, there is a lambda definable bijection $F{:}\,\mathcal{A} \to \mathcal{C}_{\mathbb{N}}$ with lambda definable inverse.

4.9. REMARK. If $\mathcal{A}$ only has a λ-definable $F{:}\,\mathcal{A} \to \mathcal{C}_{\mathbb{N}}$ injection, then $\mathcal{A}$ doesn't need to be special. Indeed, let $K \subseteq \mathbb{N}$ be re but not recursive, so that its complement $\overline{K} \subseteq \mathbb{N}$ is not re. Define $\mathcal{A} = \{\mathbf{c}_n \mid n \in \overline{K}\}$. Then $\mathsf{I}{:}\,\mathcal{A} \to \mathcal{C}_{\mathbb{N}}$ is an injection. For this $\mathcal{A}$ there is no λ-definable surjection $G{:}\,\mathcal{C}_{\mathbb{N}} \to \mathcal{A}$, for otherwise

$$\begin{array}{rcll} n \in \overline{K} & \Leftrightarrow & \mathbf{c}_n \in \mathcal{A} & \\ & \Leftrightarrow & \exists m.\mathbf{c}_n =_\beta G\mathbf{c}_m, & \quad \text{which is re,} \end{array}$$

contradicting that $\overline{K}$ is not re.

4.10. DEFINITION. $\mathcal{A}$ is called an *adequate numeral system* if there are terms $\underline{0}, \underline{S}, \underline{P}, \underline{Z_?}$ (zero, successor, predecessor, test for zero) such that, writing $\underline{n} \triangleq \underline{S}^n\underline{0}$ for $n \in \mathbb{N}$, one has

$$\begin{array}{rcl} \mathcal{A} & = & \{\underline{n} \mid n \in \mathbb{N}\}; \\ \underline{P}(\underline{n+1}) & = & \underline{n}; \\ \underline{Z_?}\,\underline{0} & = & \lambda xy.x; \\ \underline{Z_?}\,\underline{n+1} & = & \lambda xy.y. \end{array}$$

4.11. PROPOSITION. *Let $\mathcal{A} \subseteq \Lambda^{\emptyset}$ be infinite. If $\mathcal{A}$ is special, then there is a lambda definable bijection $H{:}\,\mathcal{A} \to \mathcal{C}_{\mathbb{N}}$.*

PROOF. Let combinators F, G be given as required in Definition 4.8. Define by primitive recursion

$$\begin{array}{rcll} H\mathbf{c}_0 & = & G\mathbf{c}_0; & \\ H\mathbf{c}_{n+1} & = & G\mathbf{c}_{\mu m.(G\mathbf{c}_m \notin_\beta \{H\mathbf{c}_0,\ldots,H\mathbf{c}_n\})}, & \quad (*) \end{array}$$

In (*) 'μm' stands for 'the least number such that', which in this case always exists since $\mathcal{A}$ is infinite and G surjective. That H is λ-definable follows from the existence of F: indeed, for $M, N \in \mathcal{A}$ one has

$$M \neq_\beta N \;\Leftrightarrow\; FM \neq_\beta FN \;\Leftrightarrow\; \neg Q_{=}(FM)(FN),$$

where $Q_=$ is the decidable equality predicate on Church numerals, so that also

$$G\mathbf{c}_m \notin_\beta \{H\mathbf{c}_0,\dots,H\mathbf{c}_n\} \;\Leftrightarrow\; \forall k \le n.\neg Q_=(F\circ G\mathbf{c}_m)(F\circ H\mathbf{c}_k)$$

is decidable.

Claim. For all $n\in\mathbb{N}$ one has

$$\{G\mathbf{c}_0,\dots,G\mathbf{c}_n\} \subseteq \{H\mathbf{c}_0,\dots,H\mathbf{c}_n\}.$$

The claim follows by induction on n. Case $n = 0$. By definition $H\mathbf{c}_0 = G\mathbf{c}_0$.

Case $n + 1$. Assume $\{G\mathbf{c}_0,\dots,G\mathbf{c}_n\} \subseteq \{H\mathbf{c}_0,\dots,H\mathbf{c}_n\}$ (induction hypothesis), towards $\{G\mathbf{c}_0,\dots,G\mathbf{c}_{n+1}\} \subseteq \{H\mathbf{c}_0,\dots,H\mathbf{c}_{n+1}\}$. If $G\mathbf{c}_{n+1}\in\{H\mathbf{c}_0,\dots,H\mathbf{c}_n\}$, then we are done. Otherwise $G\mathbf{c}_{n+1}\notin\{H\mathbf{c}_0,\dots,H\mathbf{c}_n\}$. For $m<(n+1)$ one has $G\mathbf{c}_m\in\{G\mathbf{c}_0,\dots,G\mathbf{c}_n\}$ which is a subset of $\{H\mathbf{c}_0,\dots,H\mathbf{c}_n\}$ by the induction hypothesis. Therefore by definition $H\mathbf{c}_{n+1} = G\mathbf{c}_{n+1}$, and the conclusion holds again. This proves the claim.

By clause (*) in the definition above H is injective. That it is also surjective follows from the claim and the surjectivity of G. ■

4.12. Corollary ([52]). *Let $\mathcal{A} \subseteq \Lambda^\varnothing$ be infinite. Then the following are equivalent.*

(i) *$\mathcal{A}$ is special.*
(ii) *$\mathcal{A}$ is separable.*
(iii) *$\mathcal{A}$ is an adequate numeral system.*

Proof. (i) ⇒ (ii). If $\mathcal{A}$ is separable, via $F{:}\,\mathcal{A}\to\mathcal{C}_\mathbb{N}$ and $G{:}\,\mathcal{C}_\mathbb{N}\to\mathcal{A}$, then by Proposition 4.11 there exists a λ-definable $H{:}\,\mathcal{C}_\mathbb{N}\to\mathcal{A}$ that is a bijection. We need to show that H has a λ-definable inverse. This $H^{-1}{:}\,\mathcal{A}\to\mathcal{C}_\mathbb{N}$ can be defined by

$$\begin{aligned} H^{-1} &= \lambda a.(\mu m.Hm =_\beta a)\\ &= \lambda a.(\mu m.F(Hm) =_\beta Fa)\\ &= \lambda a.(\mu m.Q_=(F\circ Hm)(Fa)), \qquad \text{as in the proof of the proposition.}\end{aligned}$$

(ii) ⇒ (iii). By H, H^{-1} the set $\mathcal{A}$ inherits the structure of an adequate numeral system from $\mathcal{C}_\mathbb{N}$.

(iii) ⇒ (i). Let $\underline{0},\underline{S},\underline{P},\underline{Z}_?$ give $\mathcal{A}$ the structure of an adequate numeral system. Then the computable functions can be λ-defined w.r.t. the $\underline{n}$. By primitive recursion on the $\underline{n}$ and $\mathbf{c}_n$ numerals, respectively, one can define λ-definable $F{:}\,\mathcal{A}\to\mathcal{C}_\mathbb{N}$ and $G{:}\,\mathcal{C}_\mathbb{N}\to\mathcal{A}$ satisfying $F\underline{n} = \mathbf{c}_n$ and $G\mathbf{c}_n = \underline{n}$, making $\mathcal{A}$ separable. ■

5. Translating without parsing

Combinatory terms are built-up from $\mathbf{K},\mathbf{S}$ with just application. We write all parenthesis. For example $((\mathbf{S}(\mathbf{KK}))\mathbf{S})$ is such a term. It was noticed by Corrado Böhm and Mariangiola Dezani, [14], that the meaning of such a term can be found by interpreting it symbol by symbol, including the two kinds of parentheses. One doesn't need to parse the combinator to display its tree-like structure. The method also applies to combinatory terms build from different combinators, including for example $\mathbf{B}$ corresponding to the λ-term $\mathsf{B} = \lambda fgx.f(gx) = \lambda fg.f\circ g$.

5.1. DEFINITION. Define for λ-terms M, N

$$\begin{array}{rcl} M \circ N & = & \lambda x.M(Nx); \\ M * N & = & N \circ M; \\ \langle M \rangle & = & \lambda x.xM. \end{array}$$

It is easy to see that $\circ$ and $*$ are associative modulo β-equality of the λ-calculus; moreover, for $k \geq 2$ one has

$$M_k \circ \ldots \circ M_1 \circ M_1 = \lambda x.M_k(\ldots(M_1(M_1 x))..).$$

5.2. DEFINITION. Combinatory terms $\mathcal{C}$ are built up over alphabet $\Sigma = \{\mathbf{K}, \mathbf{S}, (,)\}$ by the following context-free grammar

$$\mathcal{C} ::= \mathbf{K} \mid \mathbf{S} \mid (\mathcal{C}\,\mathcal{C})$$

5.3. DEFINITION. Given $P \in \mathcal{C}$ its translation into closed terms of the λ-calculus is P_λ defined recursively as follows:

$$\begin{array}{rcl} \mathbf{K}_\lambda & = & \mathsf{K} = \lambda xy.x; \\ \mathbf{S}_\lambda & = & \mathsf{S} = \lambda xyz.xz(yz); \\ (QR)_\lambda & = & Q_\lambda R_\lambda. \end{array}$$

For this translation the $P \in \mathcal{C}$ needs to be parsed. For example if $P = (QR)$, we need to know where the string Q ends and similarly where R starts. Böhm and Dezani found a translation that avoids this need for parsing

5.4. DEFINITION. (i) The symbols of Σ are translated into $\Lambda^\emptyset$ as follows.

$$\begin{array}{rcl} \#(& = & \mathsf{B} \\ \#\mathbf{K} & = & \langle \mathsf{K} \rangle \\ \#\mathbf{S} & = & \langle \mathsf{S} \rangle \\ \#) & = & \mathsf{I} \end{array}$$

(ii) A word in $w = a_1 \cdots a_n \in \Sigma^*$ is translated into $\varphi(w) \in \Lambda^\emptyset$ as follows.

$$\varphi(w) = \#a_1 * \cdots * \#a_n.$$

5.5. PROPOSITION. (i) *For all $P \in \mathcal{C}$ one has $\varphi(P) =_\beta \langle P_\lambda \rangle$.*
(ii) *For all $P \in \mathcal{C}$ one has $\varphi(P)\mathsf{I} =_\beta P_\lambda$.*

PROOF. (i) Since $P \in \mathcal{C}$, we may use induction over terms in $\mathcal{C}$. If $P = \mathbf{K}$ or $P = \mathbf{S}$, the result holds by definition of φ. If $P = (QR)$, then

$$\begin{array}{rcl} \varphi(P) & =_\beta & \#(*\varphi(Q) * \varphi(R) * \#),\ \text{by the associativity of } *, \\ & =_\beta & \mathsf{B} * \varphi(Q) * \varphi(R) * \mathsf{I}, \\ & =_\beta & \mathsf{I} \circ \langle Q_\lambda \rangle \circ \langle R_\lambda \rangle \circ \mathsf{B},\ \text{by definition of } * \text{ and the ind. hyp.}, \\ & =_\beta & \lambda x.\mathsf{I}(\langle Q_\lambda \rangle(\langle R_\lambda \rangle(\mathsf{B}x))), \\ & =_\beta & \lambda x.(\langle Q_\lambda \rangle(\mathsf{B}xR_\lambda)), \\ & =_\beta & \lambda x.\mathsf{B}xR_\lambda Q_\lambda, \\ & =_\beta & \lambda x.x(R_\lambda Q_\lambda) = \langle (RQ)_\lambda \rangle = \langle P_\lambda \rangle. \end{array}$$

(ii) By (i). ■

Proposition 5.5(ii) shows that the meaning of P can be obtained without parsing.

6. A simple self-evaluator

To $M \in \Lambda$ one assigns computably a Gödel-number $\#M$.

6.1. DEFINITION. For $M \in \Lambda$ its code $\ulcorner M \urcorner$ is defined as the Church numeral corresponding to $\#M$

$$\ulcorner M \urcorner \triangleq \mathbf{c}_{\#M}.$$

Note that the code of M satisfies 1. $\ulcorner M \urcorner$ is in normal form; 2. syntactic operations on M are lambda definable on $\ulcorner M \urcorner$, by the computability of $\#$. An evaluator E is constructed by Stephen Cole Kleene [35] such that for all $M \in \Lambda^{\emptyset}$ one has

$$\mathsf{E}\ulcorner M \urcorner =_{\beta} M.$$

A technical problem to define E and show this is caused by the fact that the lambda terms are inductively defined via open terms containing free variables. But the decoding only holds for closed terms. The way Kleene dealt with this (basically the problem of representing the binding effect of λx), was to translate closed λ-terms first to combinators and then representing these as numerals. The term E was reconstructed by McCarthy for the programming language LISP under the name '`eval`', and babtized by Reynolds [50] as the 'meta-circular' self-interpreter.

During lectures at Radboud University on Kleene's self-evaluator E and constructing this term via the combinators, the student Peter de Bruin came with an improvement. He suggested to use the intuition of denotational semantics of λ-calculus. First the meaning of an open term M (containing possibly free variables) is given with the use of a valuation v assigning values to free variables, $\mathsf{E}_0\ulcorner M \urcorner v$.

6.2. THEOREM (Kleene [35]). *There is a term* E *such that*

$$\forall M \in \Lambda^{\emptyset}.\mathsf{E}\ulcorner M \urcorner = M.$$

PROOF (P. de Bruin). By the effectiveness of the Gödel-numbering there exists a term E_0 satisfying

$$\begin{array}{rcl} \mathsf{E}_0\ulcorner x \urcorner v &=& v(\ulcorner x \urcorner); \\ \mathsf{E}_0\ulcorner (PQ) \urcorner v &=& (\mathsf{E}_0\ulcorner P \urcorner v)(\mathsf{E}_0\ulcorner Q \urcorner v); \\ \mathsf{E}_0\ulcorner (\lambda x.P) \urcorner v &=& \lambda y.\mathsf{E}_0\ulcorner P \urcorner (v[\ulcorner x \urcorner \mapsto y]), \end{array}$$

where $v[\ulcorner x \urcorner \mapsto y] = v'$ with

$$\begin{array}{rcll} v'\ulcorner z \urcorner &=& v\ulcorner x \urcorner, & \text{if } \ulcorner z \urcorner \neq \ulcorner x \urcorner, \\ v'\ulcorner z \urcorner &=& y, & \text{if } \ulcorner z \urcorner = \ulcorner x \urcorner. \end{array}$$

Then one can prove that for $M \in \Lambda$ with $\mathrm{FV}(M) \subseteq \{x_1, \ldots, x_n\}$ one has

$$E_0\ulcorner M \urcorner v = M[x_1, \ldots, x_n := v(\ulcorner x_1 \urcorner), \ldots, v(\ulcorner x_n \urcorner)].$$

Therefore

$$\forall M \in \Lambda^{\emptyset}.\mathsf{E}_0\ulcorner M \urcorner v = M$$

and one can take $\mathsf{E} = \lambda m.\mathsf{E}_0 m \mathsf{I}$. ■

6.3. Corollary. *The term* E *enumerates the closed* λ*-terms*

$$\forall M \in \Lambda^{\varnothing} \exists n \in \mathbb{N}. \mathsf{E}\mathbf{c}_n = M.$$

6.4. Remark. In [4] ([6]) it is proved (constructively) that any enumerator of the closed terms is reducing in the following sense.

$$\forall M \in \Lambda^{\varnothing} \exists n \in \mathbb{N}. \mathsf{E}'\mathbf{c}_n = M \;\Rightarrow\; \forall M \in \Lambda^{\varnothing} \exists n \in \mathbb{N}. \mathsf{E}'\mathbf{c}_n \twoheadrightarrow M.$$

Torben Mogensen [46] was inspired by the construction of Peter de Bruin and came up with what is called a higher order encoding of λ-terms, see [49], in which a λ is interpreted by itself.

6.5. Definition (Mogensen [46]). An open lambda term M can be interpreted as an open lambda term with the same free variables as follows.

$$\begin{array}{rcl} \ulcorner x \urcorner^m & = & \lambda abc.ax; \\ \ulcorner PQ \urcorner^m & = & \lambda abc.b\ulcorner P \urcorner^m \ulcorner Q \urcorner^m; \\ \ulcorner \lambda x.P \urcorner^m & = & \lambda abc.c(\lambda x.\ulcorner P \urcorner^m). \end{array}$$

This can be seen as first using three unspecified constructors $\mathtt{var}, \mathtt{app}, \mathtt{abs} \in \Lambda^{\varnothing}$ as follows

$$\begin{array}{rcl} \ulcorner x \urcorner^m & = & \mathsf{var}\, x; \\ \ulcorner PQ \urcorner^m & = & \mathsf{app}\, \ulcorner P \urcorner^m \ulcorner Q \urcorner^m; \\ \ulcorner \lambda x.P \urcorner^m & = & \mathsf{abs}\, (\lambda x.\ulcorner P \urcorner^m), \end{array}$$

and then taking

$$\begin{array}{rcl} \mathsf{var} & = & \lambda x \lambda abc.ax; \\ \mathsf{app} & = & \lambda pq \lambda abc.bpq; \\ \mathsf{abs} & = & \lambda z \lambda abc.cz. \end{array}$$

6.6. Theorem (Mogensen [46]). *There is an evaluator* E^m *such that for all* $M \in \Lambda$

$$\mathsf{E}^m \ulcorner M \urcorner^m = M.$$

Proof. Using Turing's fixed point combinator Θ one can construct a term E^m such that

$$E^m M \twoheadrightarrow M\mathsf{I}(B\mathsf{E}^m)(C\mathsf{E}^m),$$

where $B = \lambda epq.ep(eq)$, and $C = \lambda ezx.e(zx)$: take $\mathsf{E}^m = \Theta(\lambda em.m\mathsf{I}(Be)(Ce))$. Then by induction on the structure of $M \in \Lambda$ it follows that $\mathsf{E}^m \ulcorner M \urcorner^m \twoheadrightarrow M$.

$$\begin{array}{rcll} \mathsf{E}^m \ulcorner x \urcorner^m & \twoheadrightarrow & \ulcorner x \urcorner^m \mathsf{I}(B\mathsf{E}^m)(C\mathsf{E}^m) & \\ & \twoheadrightarrow & \mathsf{I}x \to x; & \\ \mathsf{E}^m \ulcorner PQ \urcorner^m & \twoheadrightarrow & \ulcorner PQ \urcorner^m \mathsf{I}(B\mathsf{E}^m)(C\mathsf{E}^m) & \\ & \twoheadrightarrow & B\mathsf{E}^m \ulcorner P \urcorner^m \ulcorner Q \urcorner^m & \\ & \twoheadrightarrow & \mathsf{E}^m \ulcorner P \urcorner^m (\mathsf{E}^m \ulcorner Q \urcorner^m) & \\ & \twoheadrightarrow & PQ, & \text{by the induction hypothesis;} \end{array}$$

$$
\begin{array}{rcll}
\mathsf{E}^m\ulcorner\lambda x.P\urcorner^m & \twoheadrightarrow & \ulcorner\lambda x.P\urcorner^m\mathsf{I}(B\mathsf{E}^m)(C\mathsf{E}^m) & \\
& \twoheadrightarrow & C\mathsf{E}^m(\lambda x.\ulcorner P\urcorner^m) & \\
& \twoheadrightarrow & \lambda x.\mathsf{E}^m((\lambda x.\ulcorner P\urcorner^m)x) & \\
& \to & \lambda x.\mathsf{E}^m\ulcorner P\urcorner^m & \\
& \twoheadrightarrow & \lambda x.P, & \text{by the induction hypothesis.} \blacksquare
\end{array}
$$

6.7. Remark. (i) Using Mogensen's translation, decoding is possible for all terms $M \in \Lambda$ possibly containing free variables. On the other hand not all syntactic operations are possible on the coded terms. Equality test for variables is possible for $\ulcorner x\urcorner$, but not for $\ulcorner x\urcorner^m$.

(ii) In spite of this, in [5] it is proved that for closed terms equality discrimination on coded terms $\ulcorner M\urcorner^m, \ulcorner N\urcorner^m$ is lambda definable.

(iii) In Mogensen [46] it is also proved that there is a normalizer acting on coded terms.

There is a term R^m *such that for all* $M \in \Lambda$

if M *has a normal form* N*, then* $\mathsf{R}^m\ulcorner M\urcorner^m \twoheadrightarrow \ulcorner N\urcorner^m$*;*

if M *has a no normal form, then* $\mathsf{R}^m\ulcorner M\urcorner^m$ *has no nf.*

In Berarducci-Böhm [10] a very simple self-evaluator is constructed, based on Mogensen's construction above, but using different choices for `var`, `app`, `abs`. These are based on unpublished work of Böhm and Piperno who represented algebraic data structures in such a way that primitive recursive (computable) functions are representable by terms in normal form, avoiding the fixed point operator that was used in the proof of Theorem 6.6.

6.8. Theorem (Berarducci-Böhm [10]). *There is a coding of* λ*-terms* $M \mapsto \ulcorner M\urcorner^{bb}$ *with a short closed normal form* $\mathsf{E}^{bb} = \langle\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle\rangle$ *as evaluator.*

Proof. Define

$$
\begin{array}{rcl}
\ulcorner x\urcorner^{bb} & = & \mathtt{var}^{bb}\, x; \\
\ulcorner PQ\urcorner^{bb} & = & \mathtt{app}^{bb}\, \ulcorner P\urcorner^{bb}\ulcorner Q\urcorner^{bb}; \\
\ulcorner \lambda x.P\urcorner^{bb} & = & \mathtt{abs}^{bb}\, (\lambda x.\ulcorner P\urcorner^{bb}),
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathtt{var}^{bb} & = & \lambda x\lambda e.e\mathsf{U}^3_1 xe; \\
\mathtt{app}^{bb} & = & \lambda pq\lambda e.e\mathsf{U}^3_2 pqe \\
\mathtt{abs}^{bb} & = & \lambda z\lambda e.e\mathsf{U}^3_3 ze.
\end{array}
$$

By induction on the structure of M we show that $\ulcorner M\urcorner^{bb}\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle \twoheadrightarrow M$.

Case $M = x$. Then

$$
\begin{array}{rcl}
\ulcorner x\urcorner^{bb}\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle & \twoheadrightarrow & ((\lambda x\lambda e.e\mathsf{U}^3_1 xe)x)\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle \\
& \twoheadrightarrow & (\lambda e.e\mathsf{U}^3_1 xe)\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle \\
& \twoheadrightarrow & \langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle\mathsf{U}^3_1 x\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle \\
& \twoheadrightarrow & \mathsf{K}x\langle \mathsf{K}, \mathsf{S}, \mathsf{C}\rangle \\
& \twoheadrightarrow & x.
\end{array}
$$

Case $M \equiv PQ$. Then

$$\begin{array}{rcl} \ulcorner PQ \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle & \equiv & (\lambda pqe.e\mathsf{U}_2^3 pqe) \ulcorner P \urcorner^{bb} \ulcorner Q \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \mathsf{U}_2^3 \ulcorner P \urcorner^{bb} \ulcorner Q \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \mathsf{S} \ulcorner P \urcorner^{bb} \ulcorner Q \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \ulcorner P \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle (\ulcorner Q \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle) \\ & \twoheadrightarrow & PQ, \qquad \text{by the induction hypothesis.} \end{array}$$

Case $M \equiv \lambda x.P$. Then

$$\begin{array}{rcl} \ulcorner \lambda x.P \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle & \equiv & (\lambda ze.e\mathsf{U}_3^3 ze)(\lambda x. \ulcorner P \urcorner^{bb}) \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \mathsf{U}_3^3 (\lambda x. \ulcorner P \urcorner^{bb}) \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \mathsf{C}(\lambda x. \ulcorner P \urcorner^{bb}) \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \equiv & (\lambda xyz.xzy)(\lambda x. \ulcorner P \urcorner^{bb}) \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \lambda z.(\lambda x. \ulcorner P \urcorner^{bb}) z \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \equiv & \lambda x.(\lambda x. \ulcorner P \urcorner^{bb}) x \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \to & \lambda x. \ulcorner P \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \\ & \twoheadrightarrow & \lambda x.P, \qquad \text{by the induction hypothesis.} \end{array}$$

Therefore for all $M \in \Lambda$ one has $\ulcorner M \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \twoheadrightarrow M$. It follows that

$$\mathsf{E}^{bb} \ulcorner M \urcorner^{bb} \to \ulcorner M \urcorner^{bb} \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \twoheadrightarrow M. \blacksquare$$

It is a remarkable coincidence that the term $\mathsf{E}^{bb} \equiv \langle \langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle \rangle$ represents the name "Kleene, Stephen Cole" the full name of the inventor of self-evaluation in λ-calculus. Corrado Böhm was fond of such tricks and had the nickname 'il miracolo'.

Coda

At a symposium in honor of Corrado Böhm's ninety's birthday, January 2013, at Sapienza University, Rome, the jubilee treated the audience with an open problem. Actually it is more a 'Koan' (not precisely stated) than a Problem (with a precisely stated space of answers). But Koans are often the more interesting problems in mathematics and computer science.

Problem/Koan. (C. Böhm, 2013.) Given β-normal forms $F \equiv \lambda x_1 \cdots x_n.P$, and $G \equiv \lambda x_1 \cdots x_n.Q \in \Lambda^{\emptyset}$. By writing $F^d = \lambda x.F(x\mathbf{c}_1) \ldots (x\mathbf{c}_n)$ and similarly for G^d, these terms can be made unary. Trying to find closed terms M such that $FM = GM$, what can be learned from solutions N of the equation $F^d N = G^d N$? (A *deed* is a closed nf of the form $\lambda x.xP_1 \cdots P_k$. The F^d, G^d are deeds up to $=_\beta$.)

Acknowledgments

The author thanks Marko van Eekelen for explaining him many years ago the method of bootstrapping (Section 1), Mariangiola Dezani for comments on the paper, and

Rinus Plasmeijer for discussions about Section 3. The referees provided very useful remarks, improving the paper.

To the family of Corrado Böhm I am grateful for letting me spend wonderful times with them, besides for fully enabling us to enjoy the combinators.

References

[1] E. Ashcroft and Z. Manna. The translation of goto programs into while programs. In C.V. Freiman, J.E. Griffith, and J.L. Rosenfeld, editors, *Proceedings of IFIP Congress 71*, volume 1, pages 250–255, Amsterdam, 1972. North-Holland.

[2] H. P. Barendregt. Normed uniformly reflexive structures. In *Proceedings of the Symposium on Lambda-Calculus and Computer Science Theory*, pages 272–286, Berlin, Heidelberg, 1975. Springer-Verlag.

[3] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, revised edition, 1984.

[4] H. P. Barendregt. Theoretoical Pearl: Enumerators of lambda terms are reducing. *J. of Funct. Programming*, 2(2):233–236, 1992.

[5] H. P. Barendregt. Discriminating coded lambda terms. In K.R. Apt, A.A. Schrijver, and N.M. Temme, editors, *From Universal Morphisms to Megabytes: A Baayen Space-Odyssey*, pages 141–151. CWI, 1994.

[6] H. P. Barendregt. Enumerators of lambda terms are reducing constructively. *Annals of Pure and Applied Logic*, 73:3–9, 1995.

[7] H. P. Barendregt. *Kreisel, lambda calculus, a windmill and a castle*, pages 3–14. Peters, Wellesley, Mass., 1996.

[8] H. P. Barendregt, W. J. M. Dekkers, and R. Statman. *Lambda Calculus with Types.* Perspectives in Mathematical Logic. Cambridge University Press, 2013.

[9] H. P. Barendregt, G. Manzonetto, and M. J. Plasmeijer. The imperative and functional programming paradigm. In B. Cooper and J. van Leeuwen, editors, *Alan Turing - His Work and Impact*, pages 121–126. Elsevier, 2013.

[10] A. Berarducci and C. Böhm. A self-interpreter of lambda calculus having a normal form. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, pages 85–99, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[11] C. Böhm. *Calculatrices digitales du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme.* PhD thesis, ETH, Zürich, 1954. Thesis written under supervision of E. Stiefel and P. Bernays and defended in 1951. Published in Ann. Math. PuraAppl. 37 (1954), 5-47. DOI: `doi.org/10.3929/ethz-a-000090226`.

[12] C. Böhm. The CUCH as a Formal and Description Language. In T.B. Steele Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 179–197. North Holland, 1966.

[13] C. Böhm. Alcune proprietà delle forme normali nel $\lambda\mathsf{K}$-calcolo. Technical Report 696, INAC, 1968.

[14] C. Böhm and M. Dezani. Can syntax be ignored during translation? In Nivat, editor, *Automata, Languages and Programming*, pages 197–207, 1973.

[15] C. Böhm, M. Dezani-Ciancaglini, P. Peretti, and S.Ronchi Della Rocca. A discrimination algorithm inside $\lambda\beta$-calculus. *Theoretical Computer Science*, 8(3):271 – 291, 1979.

[16] C. Böhm and W. Gross. Introduction to the CUCH. In E. R. Caianiello, editor, *Automata Theory*, pages 35–65. Academic Press, New York, 1966.

[17] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[18] A. Church. A set of postulates for the foundation of logic (1). *Annals of Mathematics*, 33:346–366, 1932.

[19] A. Church. A set of postulates for the foundation of logic (2). *Annals of Mathematics*, 34:839–864, 1933.

[20] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

[21] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[22] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[23] Clean. Pure functional language, with I/O through uniqueness types. URL: `<https://clean.cs.ru.nl/>`.

[24] D. C. Cooper. Böhm and Jacopini's reduction of flow-charts. *Comm. ACM*, 10(8):463–473, 1967.

[25] M. Coppo, M. Dezani-Ciancaglini, and S. Ronchi della Rocca. (Semi)-separability of finite sets of terms in Scott's D_∞-models of the λ-calculus. In G. Ausiello and C. Böhm, editors, *Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 142–164, Berlin, 1978. Springer.

[26] J. N. Crossley. Reminiscences of logicians. In J. N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 1–62. Springer, 1975.

[27] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Science of the USA*, 20:584–590, 1934.

[28] H. B. Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

[29] O. Dahl, E. Dijkstra, and C. Hoare, editors. *Structured Programming.* Academic Press Ltd., London, UK, 1972.

[30] E. Dijkstra. Go to statement considered harmful. *Comm. of the ACM*, 11(3):147–148, 1968.

[31] S. Guerrini, A. Piperno, and M. Dezani-Ciancaglini. *Böhm's Theorem*, chapter 1, pages 1–16. Imperial College Press, 2009. Eds. E. Gelenbe and J.-P. Kahane.

[32] D. Harel. On Folk Theorems. *Comm. of the ACM*, 23(7):379–389, 1980.

[33] Haskell. Pure functional language, with I/O through monads. URL: `<https://www.haskell.org/>`.

[34] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.

[35] S. C. Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57:153–173, 219–244, 1935.

[36] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36:630–636, 1935.

[37] D. E. Knuth. *The Art of Computer Programming*, volume 1-7. Addison Wesley Longman Publishing Co., Redwood City, CA, USA, 1968-2018. (As yet unfinished.).

[38] D. E. Knuth. Structured Programming with `go to` Statements. *Computing Surveys*, 6(4):261–301, 1974.

[39] D. E. Knuth and R.W. Floyd. Notes on avoiding `goto` statements. *Information processing letters*, 1(1):23–31, 1971.

[40] D. Kozen and Wei-Lung Dustin Tseng. The Böhm–Jacopini Theorem Is False, Propositionally. In Ph. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 177–192, Berlin, 2008. Springer.

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

[42] J. McCarthy, P.W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[43] W. M. McKeeman, J. J. Horning, and D. B. Wortman. *A compiler generator.* Automatic computation. Prentice-Hall, Englewood Cliffs, NJ, 1970.

[44] H. Mills. Mathematical foundations for structured programming. Report FSC 72-6012, IBM Federal Systems Division, Gaithersburgh, Md., 1972. 62 pp.

[45] R. Milner, M. Tofte, R. Harper, and D. McQueen. *The Definition of Standard ML*. The MIT Press, 1990.

[46] T. Æ. Mogensen. Theoretical pearls: Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1994.

[47] OCaml. derived from ML and object orientation. URL: `<ocaml.org/>`.

[48] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. Out of print. Available from URL: `<https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/>`.

[49] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[50] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, volume 2, pages 717–740. ACM, 1972.

[51] Scheme. Improved version of LISP. URL: `<groups.csail.mit.edu/mac/projects/scheme/>`.

[52] R. Statman and H. P. Barendregt. Böhm's Theorem, Church's Delta, Numeral Systems, and Ershov Morphisms. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. de Vrijer, editors, *Processes, terms and cycles: steps on the road to infinity: essays dedicated to Jan Willem Klop on the occasion of his 60th birthday*, pages 40–54. Springer, Berlin, 2005.

[53] A. M. Turing. Computability and lambda definability. *The Journal of Symbolic Logic*, 2:153–163, 1937.

[54] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.