Functional Bits: Lambda Calculus based Algorithmic Information Theory

John Tromp

April 23, 2023

Abstract

In the first part we introduce binary representations of both lambda calculus and combinatory logic, together with very concise interpreters that witness their simplicity. Along the way we present a simple graphical notation for lambda calculus, a new empty list representation, improved bracket abstraction, and a new fixpoint combinator. In the second part we review Algorithmic Information Theory, for which these interpreters provide a convenient vehicle. We demonstrate this with several concrete upper bounds on program-size complexity.

1 Introduction

The ability to represent programs as data and to map such data back to programs (known as reification and reflection [10]), is of both practical use in metaprogramming [16] as well as theoretical use in computability and logic [19]. It comes as no surprise that the pure lambda calculus, which represents both programs and data as functions, is well equipped to offer these features. In [7], Kleene was the first to propose an encoding of lambda terms, mapping them to Gödel numbers, which can in turn be represented as so called Church numerals. Decoding such numbers is somewhat cumbersome, and not particularly efficient. In search of simpler constructions, various alternative encodings have been proposed using higher-order abstract syntax [9] combined with the standard lambda representation of signatures [12]. A particularly simple encoding was proposed by Mogensen [25], for which the term $\lambda m.m(\lambda x.x)(\lambda x.x)$ acts as a selfinterpreter. The prevalent data format, both in information theory and in practice, however, is not numbers, or syntax trees, but bits. We propose binary encodings of both lambda and combinatory logic terms, and exhibit relatively simple and efficient interpreters (using the standard representation of bit-streams as lists of booleans).

This gives us a representation-neutral notion of the size of a term, measured in bits. More importantly, it provides a way to describe arbitrary data with, in a sense, the least number of bits possible. We review the notion of how a computer reading bits and outputting some result constitutes a description method, and how universal computers correspond to optimal description methods. We then pick specific universal computers based on our interpreters and prove several of the basic results of Algorithmic Information Theory with explicit constants.

2 Lambda Calculus

We only summarize the basics here. For a comprehensive treatment we refer the reader to the standard reference [20].

Assume a countably infinite set of *variables*

$$a, b, \dots, x, y, z, x_0, x_1, \dots$$

The set of lambda terms Λ is built up from variables using *abstraction*

$$(\lambda x.M)$$

and *application*

$$(M\ N),$$

where x is any variable and M, N are lambda terms. $(\lambda x.M)$ is the function that maps x to M, while $(M\ N)$ is the application of function M to argument N. We sometimes omit parentheses, understanding abstraction to associate to the right, and application to associate to the left, e.g. $\lambda x.\lambda y.x\ y\ x$ denotes $(\lambda x.(\lambda y.((x\ y)x)))$. We also join consecutive abstractions as in $\lambda x\ y.x\ y\ x$.

The free variables $FV(M)$ of a term M are those variables not bound by an enclosing abstraction. Λ^0 denotes the set of closed terms, i.e. with no free variables. The simplest closed term is the identity $\lambda x.x$.

We consider two terms *identical* if they only differ in the names of bound variables, and denote this with $\equiv$, e.g. $\lambda y.y\ x \equiv \lambda z.z\ x$. The essence of λ calculus is embodied in the β-conversion rule which equates

$$(\lambda x.M)N = M[x := N],$$

where $M[x := N]$ denotes the result of substituting N for all free occurrences of x in M (taking care to avoid variable capture by renaming bound variables in M if necessary). For example,

$$(\lambda x\ y.y\ x)y \equiv (\lambda x.(\lambda z.z\ x))y \equiv (\lambda x\ z.z\ x)y = \lambda z.z\ y.$$

A term with no β-redex, that is, no subterm of the form $(\lambda x.M)N$, is said to be in *normal form.* Terms may be viewed as denoting computations of which β-reductions form the steps, and which may halt with a normal form as the end result.

2.1 Some useful lambda terms

Define (for any $M, P, Q, \ldots, R$)

$$\begin{array}{rcl}
\mathbf{I} & \equiv & \lambda x.x \\
\mathbf{true} & \equiv & \lambda x\ y.x \\
\mathbf{nil} \equiv \mathbf{false} & \equiv & \lambda x\ y.y \\
\langle P, Q, \ldots, R\rangle & \equiv & \lambda z.z\ P\ Q \ldots\ R \\
M[0] & \equiv & M\ \mathbf{true} \\
M[i+1] & \equiv & (M\ \mathbf{false})[i] \\
\mathbf{Y} & \equiv & \lambda f.((\lambda x.x\ x)(\lambda x.f\ (x\ x))) \\
\mathbf{\Omega} & \equiv & (\lambda x.x\ x)(\lambda x.x\ x)
\end{array}$$

Note that

$$\mathbf{true}\ P\ Q = (\lambda x\ y.x)\ P\ Q = x[x := P] = P$$

$$\mathbf{false}\ P\ Q = (\lambda x\ y.y)\ P\ Q = y[y := Q] = Q,$$

justifying the use of these terms as representing the booleans.

A pair of terms like P and Q is represented by $\langle P, Q\rangle$, which allows one to retrieve its parts by applying $\langle\mathbf{true}\rangle$ or $\langle\mathbf{false}\rangle$:

$$\langle\mathbf{true}\rangle\langle P, Q\rangle = \langle P, Q\rangle\ \mathbf{true} = \mathbf{true}\ P\ Q = P$$

$$\langle\mathbf{false}\rangle\langle P, Q\rangle = \langle P, Q\rangle\ \mathbf{false} = \mathbf{false}\ P\ Q = Q.$$

Repeated pairing is the standard way of representing a sequence of terms:

$$\langle P, \langle Q, \langle R, \ldots\rangle\rangle\rangle.$$

A sequence is thus represented by pairing its first element with its *tail*—the sequence of remaining elements. The i'th element of a sequence M may be selected as $M[i]$. To wit:

$$\langle P, Q\rangle[0] = \mathbf{true}\ P\ Q = P,$$

$$\langle P, Q\rangle[i+1] \equiv (\langle P, Q\rangle\ \mathbf{false})[i] = Q[i].$$

The empty sequence, for lack of a first element, cannot be represented by any pairing, and is instead represented by **nil**. A finite sequence $P, Q, \ldots, R$ can thus be represented as $\langle P, \langle Q, \langle \ldots, \langle R, \mathbf{nil}\rangle \ldots\rangle\rangle\rangle$.

Our choice of **nil** allows for the processing of a possible empty list s with the expression

$$s\ M\ N,$$

which for $s \equiv \mathbf{nil}$ reduces to N, and for $s \equiv \langle P, Q\rangle$ reduces to $M\ P\ Q\ N$. In contrast, Barendregt [14] chose $\mathbf{I}$ to represent the empty list, which requires a more complicated list processing expression like like $s\ (\lambda a\ b\ c.c\ a\ b)\ MXN$, which for $s = \mathbf{nil}$ reduces to $N\ M\ X$, and for $s \equiv \langle P, Q\rangle$ reduces to $M\ P\ Q\ X\ N$.

$\mathbf{Y}$ is the *fixpoint* operator, that satisfies

$$\mathbf{Y} f = (\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)) = f\ (\mathbf{Y}\ f).$$

This allows one to transform a recursive definition $f = \ldots f \ldots$ into $f = \mathbf{Y}(\lambda f.(\ldots f \ldots))$, which behaves exactly as desired.

$\boldsymbol{\Omega}$ is the prime example of a term with no normal form, the equivalence of an infinite loop.

2.2 Binary strings

Binary strings are naturally represented by boolean sequences, where **true** represents 0 and **false** represents 1.

Definition 1 *For a binary string s and lambda term M, $(s : M)$ denotes the list of booleans corresponding to s, terminated with M. Thus, $(s : \mathbf{nil})$ is the standard representation of string s.*

For example, $(011 : \mathbf{nil}) \equiv \langle \mathbf{true}, \langle \mathbf{false}, \langle \mathbf{false}, \mathbf{nil}\rangle\rangle\rangle$ represents the string 011. We represent an unterminated string, such as part of an input stream, as an open term $(s : z)$, where the free variable z represents the remainder of input.

2.3 de Bruijn notation

In [13], de Bruijn proposed an alternative notation for closed lambda terms using natural numbers rather than variable names. Abstraction is simply written λM while the variable bound by the n'th enclosing λ is written as the index n. In this notation, $\lambda x\ y\ z.z\ x\ y \equiv \lambda\ \lambda\ \lambda\ 0\ 2\ 1$. It thus provides a canonical notation for all identical terms. Beta-conversion in this notation avoids variable capture, but instead requires *shifting* the indices, i.e. adjusting them to account for changes in the lambda nesting structure. Since variable/index exchanges don't affect each other, it's possible to mix both forms of notation, as we'll do later.

2.4 graphical notation

Lambda Diagrams are a graphical notation for closed lambda terms, in which abstractions are represented by horizontal lines, variables by vertical lines emanating down from their binding lambda, and applications by horizontal links connecting the leftmost variables. In the alternative style, applications link the nearest deepest variables, for a more stylistic, if less uniform, look

2.5 Binary Lambda Calculus

Definition 2 *The code for a term in de Bruijn notation is defined inductively as follows:*

$$\widehat{n} \equiv 1^{n+1}0$$

$$\begin{aligned}\widehat{\lambda M} &\equiv 00\widehat{M} \\ \widehat{M N} &\equiv 01\widehat{M}\ \widehat{N}\end{aligned}$$

We call $|\widehat{M}|$ *the* size *of* M*.*

For example $\widehat{\mathbf{I}} \equiv 0010$, $\widehat{\mathbf{false}} \equiv 000010$, $\widehat{\mathbf{true}} \equiv 0000110$ and $\widehat{\lambda x.x\ x} \equiv 00011010$, $\widehat{\lambda x.\mathbf{false}} \equiv 00000010$, of sizes 4,6,7,8 and 8 bits respectively, are the 5 smallest closed terms.

The main result of this paper is the following

Theorem 1 *There is a self-interpreter* $\mathbf{E}$ *of size 206 such that for every closed term* M *and terms* C, N *we have*

$$\mathbf{E}\ C\ (\widehat{M} : N) = C\ (\lambda z.M)\ N$$

The interpreter works in continuation passing style [17]. Given a continuation and a bitstream containing an encoded term, it returns the continuation applied to the abstracted decoded term and the remainder of the stream. The reason for the abstraction becomes evident in the proof.

The theorem is a special case of a stronger one that applies to arbitrary de Bruijn terms. Consider a de Bruijn term M in which an index n occurs at a depth of $i \leq n$ nested lambda's. E.g., in $M \equiv \lambda 3$, the index 3 occurs at depth 1. This index is like a free variable in that it is not bound within M. The interpreter (being a closed term) applied to other closed terms, cannot produce anything but a closed term. So it cannot possibly reproduce M. Instead, it produces terms that expect a list of bindings for free indices. These take the form $M^{z[]}$, which is defined as the result of replacing every free index in M, say n at depth $i \leq n$, by $z[n-i]$. For example, $(\lambda 3)^{z[]} = \lambda z[3-1] = \lambda(z\ \mathbf{false}\ \mathbf{false}\ \mathbf{true})$, selecting binding number 2 from binding list z.

The following claim (using mixed notation) will be needed later.

Claim 1 *For any de Bruijn term* M*, we have* $(\lambda M)^{z[]} = \lambda y.M^{\langle y,z\rangle []}$

Proof: A free index n at depth $i \leq n$ in M, gets replaced by $\langle y, z\rangle[n-i]$ on the right. If $i < n$ then n is also free in λM at depth $i+1$ and gets replaced by $z[n-i-1] = \langle y, z\rangle[n-i]$. If $i = n$ then n is bound by the front λ, while $\langle y, z\rangle[n-i] = \langle y, z\rangle[0] = y$.

To prove Theorem 1 it suffices to prove the more general:

Theorem 2 *There is a self-interpreter* $\mathbf{E}$ *of size 206, such that for all terms* M, C, N *we have*

$$\mathbf{E}\ C\ (\widehat{M} : N) = C\ (\lambda z.M^{z[]})\ N$$

Proof: We take

$$\begin{array}{rcl}
\mathbf{E} & \equiv & \mathbf{Y}\ (\lambda e\ c\ s.s\ (\lambda a\ t.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1))) \\
\mathbf{E}_0 & \equiv & e\ (\lambda x.b\ (c\ (\lambda z\ y.x\ \langle y, z\rangle))(e\ (\lambda y.c\ (\lambda z.x\ z\ (y\ z))))) \\
\mathbf{E}_1 & \equiv & (b\ (c\ (\lambda z.z\ b))(\lambda s.e\ (\lambda x.c\ (\lambda z.x\ (z\ b)))\ t))
\end{array}$$

of size 217 and note that the beta reduction from $\mathbf{Y}\ M$ to $(\lambda x.x\ x)(\lambda x.M\ (x\ x))$ saves 7 bits, while[15] replacing $\mathbf{E}$ by $\mathbf{Y}\ (\lambda e\ c\ s.s\ (\lambda a\ t\ e\ c.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1))ec)$ and doing one more beta reduction of $(\lambda x.M\ (x\ x))$, saves another 4 bits.

Recall from the discussion of $\mathbf{Y}$ that the above is a transformed recursive definition where e will take the value of $\mathbf{E}$.

Intuitively, $\mathbf{E}$ works as follows. Given a continuation c and sequence s, it extracts the leading bit a and tail t of s, extracts the next bit b, and selects $\mathbf{E}_0$ to deal with $a = \mathbf{true}$ (abstraction or application), or $\mathbf{E}_1$ to deal with $a = \mathbf{false}$ (an index).

$\mathbf{E}_0$ calls $\mathbf{E}$ recursively, extracting a decoded term x. In case $b = \mathbf{true}$ (abstraction), it prepends a new variable y to bindings list z, and returns the continuation applied to the decoded term provided with the new bindings. In case $b = \mathbf{false}$ (application), it calls $\mathbf{E}$ recursively again, extracting another decoded term y, and returns the continuation applied to the application of the decoded terms provided with shared bindings.

$\mathbf{E}_1$, in case $b = \mathbf{true}$, decodes to the 0 binding selector. In case $b = \mathbf{false}$, it calls $\mathbf{E}$ recursively on t (coding for an index one less) to extract a binding selector x, which is provided with the tail $z\ b$ of the binding list to obtain the correct selector.

We continue with the formal proof, using induction on M.

Consider first the case where $M = 0$. Then

$$\begin{array}{rcl}
\mathbf{E}\ C\ (\widehat{M} : N) & = & \mathbf{E}\ C\ (10 : N) \\
 & = & \langle \mathbf{false}, \langle \mathbf{true}, N\rangle\rangle\ (\lambda a\ t.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1)) \\
 & = & \langle \mathbf{true}, N\rangle\ (\lambda b.\mathbf{false}\ \mathbf{E}_0\ \mathbf{E}_1) \\
 & = & (\mathbf{E}_1\ N)[b := \mathbf{true}] \\
 & = & C\ (\lambda z.z\ \mathbf{true})\ N,
\end{array}$$

as required. Next consider the case where $M = n + 1$. Then, by induction,

$$\begin{array}{rcl}
\mathbf{E}\ C\ (\widehat{M} : N) & = & \mathbf{E}\ C\ (1^{n+2}0 : N) \\
 & = & \langle \mathbf{false}, \langle \mathbf{false}, (1^n 0 : N)\rangle\rangle\ (\lambda a\ t.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1)) \\
 & = & (\lambda s.e\ (\lambda x.C\ (\lambda z.x\ (z\ \mathbf{false})))(1^{n+1}0 : N))(1^n 0 : N) \\
 & = & \mathbf{E}\ (\lambda x.C\ (\lambda z.x\ (z\ \mathbf{false})))\ (\widehat{n} : N) \\
 & = & (\lambda x.C\ (\lambda z.x\ (z\ \mathbf{false})))\ (\lambda z.n^{z[]})\ N \\
 & = & C\ (\lambda z.n^{(z\ \mathbf{false})[]})\ N \\
 & = & C\ (\lambda z.(z\ \mathbf{false})[n]))\ N \\
 & = & C\ (\lambda z.z[n+1]))\ N \\
 & = & C\ (\lambda z.(n+1)^{z[]})\ N,
\end{array}$$

as required. Next consider the case $M = \lambda M'$. Then, by induction and claim 1,

$$\begin{aligned}
\mathbf{E}\ C\ ((\widehat{\lambda M'} : N)) &= \mathbf{E}\ C\ (00\widehat{M'} : N) \\
&= \langle \mathbf{true}, \langle \mathbf{true}, (\widehat{M'} : N)\rangle\rangle\ (\lambda a\ t.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1)) \\
&= e\ (\lambda x.(C\ (\lambda z\ y.x\langle y, z\rangle)))\ (\widehat{M'} : N) \\
&= (\lambda x.(C\ (\lambda z\ y.x\ \langle y, z\rangle)))(\lambda z.M'^{z[]})\ N \\
&= C\ (\lambda z\ y.(\lambda z.M'^{z[]})\ \langle y, z\rangle)\ N \\
&= C\ (\lambda z.(\lambda y.M'^{\langle y,z\rangle[]}))\ N \\
&= C\ (\lambda z.(\lambda M')^{z[]}))\ N,
\end{aligned}$$

as required. Finally consider the case $M = M'\ M''$. Then, by induction,

$$\begin{aligned}
\mathbf{E}\ C\ (\widehat{M'\ M''} : N) &= \mathbf{E}\ C\ (01\widehat{M'}\ \widehat{M''} : N) \\
&= \langle \mathbf{true}, \langle \mathbf{false}, (\widehat{M'}\ \widehat{M''} : N)\rangle\rangle(\lambda a\ t.t\ (\lambda b.a\ \mathbf{E}_0\ \mathbf{E}_1)) \\
&= e\ (\lambda x.(e\ (\lambda y.C\ (\lambda z.x\ z\ (y\ z)))))\ (\widehat{M'}\ \widehat{M''} : N) \\
&= (\lambda x.(e\ (\lambda y.C\ (\lambda z.x\ z\ (y\ z)))))(\lambda z.M'^{z[]})\ (\widehat{M''} : N) \\
&= e\ (\lambda y.C\ (\lambda z.(\lambda z.M'^{z[]})\ z\ (y\ z)))(\widehat{M''} : N) \\
&= (\lambda y.C\ (\lambda z.M'^{z[]}\ (y\ z)))(\lambda z.M''^{z[]})\ N \\
&= C\ (\lambda z.M'^{z[]}\ M''^{z[]})\ N \\
&= C\ (\lambda z.(M'\ M'')^{z[]})\ N,
\end{aligned}$$

as required. This completes the proof of Theorem 1.

We conjecture that any self-interpreter for any binary representation of lambda calculus must be at least 24 bytes in size, which would make E close to optimal.

3 Combinatory Logic

Combinatory Logic (CL) is the equational theory of *combinators*—terms built up, using application only, from the two constants $\mathbf{K}$ and $\mathbf{S}$, which satisfy

$$\begin{aligned}
\mathbf{S}\ M\ N\ L &= M\ L\ (N\ L) \\
\mathbf{K}\ M\ N &= M
\end{aligned}$$

CL may be viewed as a subset of lambda calculus, in which $\mathbf{K} \equiv \lambda x\ y.x$, $\mathbf{S} \equiv \lambda x\ y\ z.x\ z\ (y\ z)$, and where the beta conversion rule can only be applied groupwise, either for an $\mathbf{S}$ with 3 arguments, or for a $\mathbf{K}$ with 2 arguments. Still, the theories are largely the same, becoming equivalent in the presence of the rule of extensionality (which says $M = M'$ if $M\ N = M'\ N$ for all terms N).

A process known as *bracket abstraction* allows for the translation of any lambda term to a *combination*—a CL term containing variables in addition to

K and **S**. It is based on the following identities, which are easily verified:

$$\begin{array}{rcll} \lambda x.x = \mathbf{I} & = & \mathbf{S}\ \mathbf{K}\ \mathbf{K} & \\ \lambda x.M & = & \mathbf{K}\ M & (x \text{ not free in } M) \\ \lambda x.M\ N & = & \mathbf{S}\ (\lambda x.M)\ (\lambda x.N) & \end{array}$$

λ's can thus be successively eliminated, e.g.:

$$\begin{array}{rcl} \lambda x\ y.y\ x & \equiv & \lambda x\ (\lambda y.y\ x) \\ & = & \lambda x\ (\mathbf{S}\ \mathbf{I}(\mathbf{K}\ x)) \\ & = & \mathbf{S}\ (\mathbf{K}\ (\mathbf{S}\ \mathbf{I}))(\mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{I}), \end{array}$$

where **I** is considered a shorthand for **S K K**.

Bracket abstraction is an operation λ^0 on combinations M with respect to a variable x, such that the resulting combination contains no occurrence of x and behaves as $\lambda x.M$:

$$\begin{array}{rcll} \lambda^0 x.\ x & \equiv & \mathbf{I} & \\ \lambda^0 x.\ M & \equiv & \mathbf{K}\ M & (x \notin M) \\ \lambda^0 x.\ (M\ N) & \equiv & \mathbf{S}\ (\lambda^0 x.\ M)\ (\lambda^0 x.\ N) & \end{array}$$

3.1 Binary Combinatory Logic

Combinators have a wonderfully simple encoding as binary strings: encode **S** as 00, **K** as 01, and application as 1.

Definition 3 *We define the encoding $\widetilde{C}$ of a combinator C as*

$$\begin{array}{rcl} \widetilde{\mathbf{K}} & \equiv & 00 \\ \widetilde{\mathbf{S}} & \equiv & 01 \\ \widetilde{C\ D} & \equiv & 1\ \widetilde{C}\ \widetilde{D} \end{array}$$

Again we call $|\widetilde{C}|$ the size *of combinator C.*

For instance, the combinator $\mathbf{S}(\mathbf{KSS}) \equiv (\mathbf{S}((\mathbf{KS})\mathbf{S}))$ is encoded as 1 01 1 1 00 01 01. The size of a combinator with n $\mathbf{K}/\mathbf{S}$'s, which necessarily has $n-1$ applications, is thus $2n + n - 1 = 3n - 1$.

For such a simple language we expect a similarly simple interpreter.

Theorem 3 *There is a cross-interpreter* **F** *of size 119, such that for every combinator M and terms C, N we have*

$$\mathbf{F}\ C\ (\widetilde{M} : N) = C\ M\ N$$

Proof: We take

$$
\begin{array}{rcl}
\mathbf{F} & \equiv & \mathbf{Y}\ (\lambda e\ c\ s.s(\lambda a.a\ \mathbf{F}_0\ \mathbf{F}_1)) \\
\mathbf{F}_0 & \equiv & \lambda t.t\ (\lambda b.c\ (b\ \mathbf{K}\ \mathbf{S})) \\
\mathbf{F}_1 & \equiv & e\ (\lambda x.e\ (\lambda y.(c\ (x\ y))))
\end{array}
$$

of size 131 and note that a toplevel beta reduction saves 7 bits in size, while replacing **K** by b saves another 5 bits (we don't define **F** that way because of its negative impact on bracket abstraction).

Given a continuation c and sequence s, it extracts the leading bit a of s, and tail t extracts the next bit b, and selects $\mathbf{F}_0$ to deal with $a = \mathbf{true}$ (**K** or **S**), or $\mathbf{F}_1$ to deal with $a = \mathbf{false}$ (application). Verification is straightforward and left as an exercise to the reader.

We conjecture that any self-interpreter for any binary representation of combinatory logic must be at least 14 bytes in size. The next section considers translations of F which yield a self-interpreter of CL.

3.2 Improved bracket abstraction

The basic form of bracket abstraction is not particularly efficient. Applied to **F**, it produces a combinator of size 536.

A better version is λ^1, which uses the additional rule

$$\lambda^1 x.\ (M\ x) \equiv M \quad (x \notin M)$$

whenever possible. Now the size of **F** as a combinator is only 281, just over half as big.

Turner [26] noticed that repeated use of bracket abstraction can lead to a quadratic expansion on terms such as

$$\mathbf{X} \equiv \lambda a\ b\ \ldots\ z.(a\ b\ \ldots\ z)\ (a\ b\ \ldots\ z),$$

and proposed new combinators to avoid such behaviour. We propose to achieve a similar effect with the following set of 9 rules in decreasing order of applicability:

$$
\begin{array}{rcl}
\lambda^2 x.\ (\mathbf{S}\ \mathbf{K}\ M) & \equiv & \mathbf{S}\ \mathbf{K}\ \ \text{(for all } M\text{)} \\
\lambda^2 x.\ M & \equiv & \mathbf{K}\ M \quad (x \notin M) \\
\lambda^2 x.\ x & \equiv & \mathbf{I} \\
\lambda^2 x.\ (M\ x) & \equiv & M \quad (x \notin M) \\
\lambda^2 x.\ (x\ M\ x) & \equiv & \lambda^2 x.\ (\mathbf{S}\ \mathbf{S}\ \mathbf{K}\ x\ M) \\
\lambda^2 x.\ (M\ (N\ L)) & \equiv & \lambda^2 x.\ (\mathbf{S}\ (\lambda^2 x.\ M)\ N\ L)\ \ (M, N \text{ combinators}) \\
\lambda^2 x.\ ((M\ N)\ L) & \equiv & \lambda^2 x.\ (\mathbf{S}\ M\ (\lambda^2 x.\ L)\ N)\ \ (M, L \text{ combinators}) \\
\lambda^2 x.\ ((M\ L)\ (N\ L)) & \equiv & \lambda^2 x.\ (\mathbf{S}\ M\ N\ L)\ \ (M, N \text{ combinators}) \\
\lambda^2 x.\ (M\ N) & \equiv & \mathbf{S}\ (\lambda^2 x.\ M)\ (\lambda^2 x.\ N)
\end{array}
$$

The first rule exploits the fact that $\mathbf{S}\ \mathbf{K}\ M$ behaves as identity, whether M equals $\mathbf{K}, x$ or anything else. The fifth rule avoids introduction of two $\mathbf{I}s$. The sixth rule prevents occurrences of x in L from becoming too deeply nested, while the seventh does the same for occurrences of x in N. The eighth rule abstracts an entire expression L to avoid duplication. The operation $\lambda^2 x.\ M$ for combinators M will normally evaluate to $\mathbf{K}\ M$, but takes advantage of the first rule by considering any $\mathbf{S}\ \mathbf{K}\ M$ a combinator. Where λ^1 gives an $\mathbf{X}$ combinator of size 2030, λ^2 brings this down to 374 bits.

For $\mathbf{F}$ the improvement is more modest, to 275 bits. For further improvements we turn our attention to the unavoidable fixpoint operator.

$\mathbf{Y}$, due to Curry, is of minimal size in the λ calculus. At 25 bits, it's 5 bits shorter than Turing's alternative fixpoint operator

$$\mathbf{Y}' \equiv (\lambda z.z\ z)(\lambda z.\lambda f.f\ (z\ z\ f)).$$

But these translate to combinators of size 65 and 59 bits respectively.

In comparison, the fixpoint operator

$$\mathbf{Y}'' \equiv (\lambda x\ y.x\ y\ x)(\lambda y\ x.y(x\ y\ x))$$

translates to combinator

$$\mathbf{S}\ \mathbf{S}\ \mathbf{K}\ (\mathbf{S}\ (\mathbf{K}\ (\mathbf{S}\ \mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ \mathbf{S}\ \mathbf{K}))))\ \mathbf{K})$$

of size 35, the smallest possible fixpoint combinator as verified by exhaustive search by computer.

(The situation is similar for $\mathbf{\Omega}$ which yields a combinator of size 41, while $\mathbf{S}\ \mathbf{S}\ \mathbf{K}\ (\mathbf{S}\ (\mathbf{S}\ \mathbf{S}\ \mathbf{K}))$, of size 20, is the smallest *unsolvable* combinator—the equivalent of an undefined result, see [20]).

Using $\mathbf{Y}''$ instead of $\mathbf{Y}$ gives us the following

Theorem 4 *There is a self-interpreter* $\mathbf{F}$ *for Combinatory Logic of size 263.*

Comparing theorems 3 and 4, we conclude that λ-calculus is a much more concise language than CL. Whereas in binary λ-calculus, an abstraction takes only 2 bits plus $i+1$ bits for every occurrence of the variable at depth i, in binary CL the corresponding bracket abstraction typically introduces at least one, and often several $\mathbf{S}$'s and $\mathbf{K}$'s (2 bits each) per level of depth per variable occurrence.

4 Program Size Complexity

Intuitively, the amount of information in an object is the size of the shortest program that outputs the object. The first billion digits of π for example, contain little information, since they can be calculated by a program of a few lines only. Although information content may seem to be highly dependent on choice of programming language, the notion is actually invariant up to an additive constant.

The theory of program size complexity, which has become known as *Algorithmic Information Theory* or *Kolmogorov complexity* after one of its founding fathers, has found fruitful application in many fields such as combinatorics, algorithm analysis, machine learning, machine models, and logic.

In this section we propose a concrete definition of Kolmogorov complexity that is (arguably) as simple as possible, by turning the above interpreters into a 'universal computer'.

Intuitively, a computer is any device that can read bits from an input stream, perform computations, and (possibly) output a result. Thus, a computer is a method of description in the sense that the string of bits read from the input describes the result. A universal computer is one that can emulate the behaviour of any other computer when provided with its description. Our objective is to define, concretely, for any object x, a measure of complexity of description $C(x)$ that shall be the length of its shortest description. This requires fixing a description method, i.e. a computer. By choosing a universal computer, we achieve invariance: the complexity of objects is at most a constant greater than under any other description method.

Various types of computers have been considered in the past as description methods.

Turing machines are an obvious choice, but turn out to be less than ideal: The operating logic of a Turing machine—its *finite control*—is of an irregular nature, having no straightforward encoding into a bitstring. This makes construction of a universal Turing machine that has to parse and interpret a finite control description quite challenging. Roger Penrose takes up this challenge in his book [1], at the end of Chapter 2, resulting in a universal Turing machine whose own encoding is an impressive 5495 bits in size, over 26 times that of **E**.

The ominously named language 'Brainfuck' which advertises itself as "An Eight-Instruction Turing-Complete Programming Language" [24], can be considered a streamlined form of Turing machine. Indeed, Oleg Mazonka and Daniel B. Cristofani [18] managed to write a very clever BF self-interpreter of only 423 instructions, which translates to $423 \cdot \log(8) = 1269$ bits (the alphabet used is actually ASCII at 7 or 8 bits per symbol, but the interpreter could be redesigned to use 3-bit symbols and an alternative program delimiter).

In [5], Levin stresses the importance of a (descriptional complexity) measure, which, when compared with other natural measures, yields small constants, of at most a few hundred bits. His approach is based on *constructive objects* (c.o.'s) which are functions from and to lower ranked c.o.'s. Levin stops short of exhibiting a specific universal computer though, and the abstract, almost topological, nature of algorithms in the model complicates a study of the constants achievable.

In [2], Gregory Chaitin paraphrases John McCarthy about his invention of LISP, as "This is a better universal Turing machine. Let's do recursive function theory that way!" Later, Chaitin continues with "So I've done that using LISP because LISP is simple enough, LISP is in the intersection between theoretical and practical programming. Lambda calculus is even simpler and more elegant than LISP, but it's unusable. Pure lambda calculus with combinators S and K, it's beautifully elegant, but you can't really run programs that way, they're too slow."

There is however nothing intrinsic to λ calculus or CL that is slow; only such choices as Church numerals for arithmetic can be said to be slow, but one is free to do arithmetic in binary rather than in unary. Frandsen and Sturtivant [11] amply demonstrate the efficiency of λ calculus with a linear time implementation of k-tree Turing Machines. Clear semantics should be a primary concern, and Lisp is somewhat lacking in this regard [4]. This paper thus develops the approach suggested but discarded by Chaitin.

4.1 Functional Complexity

By providing the appropriate continuations to the interpreters that we constructed, they become universal computers describing functional terms modulo equality. Indeed, for

$$\begin{aligned} \mathbf{U} &\equiv \mathbf{E}\ \langle \mathbf{\Omega} \rangle \\ \mathbf{U'} &\equiv \mathbf{F}\ \mathbf{I} \end{aligned}$$

of sizes $|\widehat{\mathbf{U}}| = 232$ and $|\widetilde{\mathbf{U'}}| = 272$, Theorems 1 and 3 give

$$\begin{aligned} \mathbf{U}\ (\widehat{M} : N) &= M\ N \\ \mathbf{U'}\ (\widetilde{M} : N) &= M\ N \end{aligned}$$

for every closed λ-term or combinator M and arbitrary N, immediately establishing their universality.

The universal computers essentially define new binary languages, which we may call *universal binary lambda calculus* and *universal combinatory logic*, whose programs comprise two parts. The first part is a program in one of the original binary languages, while the second part is all the binary data that is consumed when the first part is interpreted. It is precisely this ability to embed arbitrary binary data in a program that allows for universality.

Note that by Theorem 2, the continuation $\langle \mathbf{\Omega} \rangle$ in U results in a term $M^{\mathbf{\Omega}[]}$. For closed M, this term is identical to M, but in case M is not closed, a free index n at λ-depth n is now bound to $\mathbf{\Omega}[n-n]$, meaning that any attempt to apply free indices diverges. Thus the universal computer essentially forces programs to be closed terms.

We can now define the Kolmogorov complexity of a term x, which comes in four flavors. In the *simple* version, programs are terminated with $N = \mathbf{nil}$ and the result must equal x. In the *prefix* version, programs are terminated with unsolvable Ω, and the result must x. In a former prefix definition, now considered unnecessarily complicated and renamed Kp, programs were not terminated, and the result had to equal the pair of x and the remainder of the input. In all cases the complexity is conditional on zero or more terms y_i.

Definition 4

$$KS(x|y_1,\ldots,y_k) = \min\{l(p) \mid \mathbf{U}\ (p : \mathbf{nil})\ y_1\ \ldots\ y_k = x\}$$

$$
\begin{aligned}
KP(x|y_1,\ldots,y_k) &= \min\{l(p) \mid \mathbf{U}\ (p:\Omega)\ y_1\ \ldots\ y_k = x\} \\
Kp(x|y_1,\ldots,y_k) &= \min\{l(p) \mid \mathbf{U}\ (p:z)\ y_1\ \ldots\ y_k = \langle x,z\rangle\}
\end{aligned}
$$

The definition also applies to infinite terms according to the *infinitary lambda calculus* of [21].

In the special case of $k = 0$ we obtain the unconditional complexities $KS(x)$ and $KP(x)$.

Finally, for a binary string s, we can define its *monotone* complexity as

$$KM(s|y_1,\ldots,y_k) = \min\{l(p) \mid \exists M : \mathbf{U}\ (p:\boldsymbol{\Omega})\ y_1\ \ldots\ y_k = (s:M)\}.$$

In this version, we consider the partial outputs produced by increasingly longer prefixes of the input, and the complexity of s is the shortest program that causes the output to have prefix s.

4.2 Monadic IO

The reason for preserving the remainder of input in the prefix casse is to facilitate the processing of concatenated descriptions, in the style of monadic IO [22]. Although a pure functional language like λ calculus cannot define functions with side effects, as traditionally used to implement IO, it can express an abstract data type representing IO actions; the IO monad. In general, a monad consists of a type constructor and two functions, *return* and *bind* (also written `>>=` in infix notation) which need to satisfy certain axioms [22]. IO actions can be seen as functions operating on the whole state of the world, and returning a new state of the world. Type restrictions ensure that IO actions can be combined only through the bind function, which according to the axioms, enforces a sequential composition in which the world is single-threaded. Thus, the state of the world is never duplicated or lost. In our case, the world of the universal machine consists of only the input stream. The only IO primitive needed is **readBit**, which maps the world onto a pair of the bit read and the new world. But a list is exactly that; a pair of the first element and the remainder. So **readBit** is simply the identity function! The **return** function, applied to some x, should map the world onto the pair of x and the unchanged world, so it is defined by **return** $\equiv \lambda x\ y.\langle x,y\rangle$. Finally, the bind function, given an action x and a function f, should subject the world y to action x (producing some $\langle a,y'\rangle$) followed by action fa, which is defined by **bind** $\equiv \lambda x\ f\ y.x\ y\ f$ (note that $\langle a,y'\rangle f = f\ a\ y'$) One may readily verify that these definitions satisfy the monad axioms. Thus, we can wite programs for U either by processing the input stream explicitly, or by writing the program in monadic style. The latter can be done in the pure functional language 'Haskell' [23], which is essentially typed lambda calculus with a lot of syntactic sugar.

4.3 An Invariance Theorem

The following theorem is the first concrete instance of the Invariance Theorem, 2.1.1 in [6].

Theorem 5 *Define $KS'(x|y_1, \ldots, y_k)$ and $KP'(x|y_1, \ldots, y_k)$ analogous to Definition 4 in terms of $\mathbf{U}'$. Then $KS(x) \leq KS'(x) + 125$ and $KP(x) \leq KP'(x) + 125$.*

The proof is immediate from Theorem 3 by using $\widehat{\mathbf{U}'}$ of length 125 as prefix to any program for $\mathbf{U}'$. We state without proof that a redesigned $\mathbf{U}$ translates to a combinator of size 617, which thus forms an upper bound in the other direction.

Now that complexity is defined for as rich a class of objects as terms (modulo equality), it is easy to extend it to other classes of objects by mapping them into λ terms.

For binary strings, this means mapping string s onto the term $(s : \mathbf{nil})$. And for a tuple of binary strings $s_0, \ldots, s_k$, we take $\langle (s_0 : \mathbf{nil}), \ldots, (s_k : \mathbf{nil}) \rangle$.

We next look at numbers in more detail, revealing a link with self-delimiting strings.

4.4 Numbers and Strings

Consider the following correspondence between natural numbers, binary strings, and strings over $\{1, 2\}$:

$n \in \mathbb{N}$:	0	1	2	3	4	5	6	7	8	9	...
$x \in \{0,1\}^*$:	ϵ	0	1	00	01	10	11	000	001	010	...
$y \in \{1,2\}^*$:	ϵ	1	2	11	12	21	22	111	112	121	...

in which the number n corresponds to

- the n-th binary string $x$in lexicographic order
- the string x obtained by stripping the leading 1 from $(n + 1)_2$, the binary representation of $n + 1$
- the string $y = n_{\{1,2\}}$, the base 2 positional system using digits $\{1, 2\}$.

Indeed, $n+1 = 2^l + \sum_{i=0}^{l-1} x_i 2^i$ iff $n = \sum_{i=0}^{l-1} 2^i + \sum_{i=0}^{l-1} x_i 2^i$ iff $n = \sum_{i=0}^{l-1}(x_i + 1)2^i$.

4.5 Prefix codes

Another way to tie the natural numbers and binary strings together is by what we shall call the *binary natural tree* shown in Figure 1. It has the set of natural numbers as vertices, and the set of binary strings as edges, such that the 2^n length-n strings are the edges leading from vertex n. Edge w leads from vertex $|w|$ to $w + 1$, which in binary is $1w$.

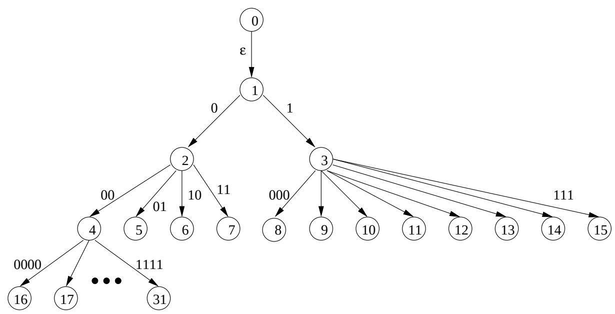

Figure 1: binary natural tree

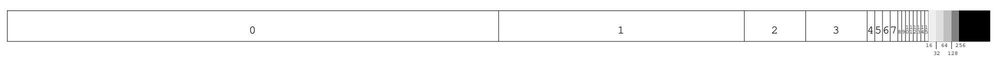

Figure 2: codes on the unit interval; $\overline{0} = 0, \overline{1} = 10, \overline{2} = 110\ 0, \overline{3} = 110\ 1, \overline{4} = 1110\ 0\ 00, \overline{5} = 1110\ 0\ 01, \overline{6} = 1110\ 0\ 10, \overline{7} = 1110\ 0\ 11, \overline{8} = 1110\ 1\ 000, etc.$.

In [8], Vladimir Levenshtein defines a universal code for the natural numbers that corresponds to concatenating the edges on the path from 0 to n, prefixed with a unary encoding of the depth of vertex n in the tree. The resulting set of codewords is *prefix-free*. meaning that no string is a proper prefix of another, which is the same as saying that the strings in the set are self-delimiting. Prefix-free sets satisfy the *Kraft inequality*: $\sum_s 2^{-|s|} \leq 1$. We've already seen two important examples of prefix-free sets, namely the set of λ term encodings $\widehat{M}$ and the set of combinator encodings $\widetilde{M}$. The Levenshtein code $\overline{n}$ of a number n can be defined recursively as

$$\overline{0} = 0 \qquad \overline{n+1} = 1\ \overline{l(n)}\ n.$$

and satisfies the following nice properties:

- prefix-free and complete: $\sum_{n \geq 0} 2^{-|\overline{n}|} = 1$.
- identifies lexicographic with numeric order: $\overline{m}$ lexicographically precedes $\overline{n}$ if and only if $m < n$.
- simple to encode and decode
- efficient in that for every k: $|\overline{n}| \leq l(n) + l(l(n)) + \cdots + l^{k-1}(n) + O(l^k(n))$, where $l(s)$ denotes the length of a string s.

Figure 2 shows the codes as segments of the unit interval, where code x covers all the real numbers whose binary expansion starts as $0.x$, and lexicographic order translates into left-to-right order.

5 Upper bounds on complexity

Having provided concrete definitions of all key ingredients of algorithmic information theory, it is time to prove some concrete results about the complexity of strings.

The simple complexity of a string is upper bounded by its length:

$$KS(x) \leq |\widehat{\mathbf{I}}| + l(x) = l(x) + 4$$

The prefix complexity of a string is upper bounded by the length of its delimited version:

$$KP(x) \leq |\widehat{\mathbf{delimit}}| + l(\overline{x}) = l(\overline{x}) + 326.$$

where **delimit** is an optimized translation of the following code

```
let
 id = \x x;
 nil = \x\y y;
 -- readbit cont church_(n+1) list returns cont church_(n_list+1)
 -- where n_list is the number corresponding to the concatenation of n and list
 readbit = \cont\n\list list (\bit cont (\f\x (n f (n f (bit x (f x))))));
 -- dlmt cont list returns cont church_n rest_list
 dlmt = \cont\list list (\bit bit (cont nil) (dlmt (\len len readbit cont id)));
 -- incc cont done list return either cont (if carry) or done (if no carry) of incremented list
 incc = \cont\done\list list (\msb incc (\r\_ msb done cont (\z z (\x\y msb y x) r))
                                        (\r\_     done      (\z z       msb      r))
                             ) (cont list);
 -- inc list returns incremented list
 inc = incc (\r\z z (\x\y x) r) id;
in dlmt (\n\l n inc nil)
```

The prefix complexity of a pair is upper bounded by the sum of individual prefix complexities, one of which is conditional on the shortest program of the other:

$$KP(x, y) \leq KP(x) + KP(y|x^*) + 784.$$

This is the easy side of the fundamental “Symmetry of information” theorem $KP(x) - KP(x|y^*) = KP(y) - KP(y|x^*) + O(1)$, which says that y contains as much information about x as x does about y.

In [3], Chaitin proves the same theorem using a resource bounded evaluator, which in his version of LISP comes as a primitive called ”try”. His proof is embodied in the program gamma:

```
((’ (lambda (loop) ((’ (lambda (x*) ((’ (lambda (x) ((’ (lambda (y) (cons x
(cons y nil)))) (eval (cons (’ (read-exp)) (cons (cons ’ (cons x* nil))
nil)))))) (car (cdr (try no-time-limit (’ (eval (read-exp))) x*)))))) (loop
nil)))) (’ (lambda (p) (if(= success (car (try no-time-limit (’ (eval
(read-exp))) p))) p (loop (append p (cons (read-bit) nil))))))
```

of length 2872 bits.

We constructed an equivalent of "try" from scratch. The constant 784 is the size of the term `pairup` defined below, containing a monadic lambda calculus interpreter that tracks the input bits read so far (which due to space restrictions is only sparsely commented):

```
\io. let
  id = \x x;
  true = \x \y x;
  false = \x \y y;
  nil = false;
  -- parse binary lambda calculus using HOAS, capturing program.
  -- uni :: ((t -> t) -> t)      -- abstraction
  --     -> (t -> (t -> t))      -- application
  --     -> (([t] -> t) -> ([Bool] -> [Bool]) -> [Bool] -> r)
  --        -- continuation taking program, parsed string, remainder of input
  --     -> ([Bool] -> [Bool])  -- initial difference list (id)
  --     -> [Bool]              -- input
  --     -> r
  uni = \abs\app.let uni0 = \cnt\ps\xs.
    xs (\b0.let ps0 = \ts.ps (\p.p b0 ts) in
      \ys\uni0\cnt.ys (\b1.
        let ps1 = \ts.ps0 (\p.p b1 ts) in
          b0 (uni0 (\v1.(b1 (cnt (\ctx.abs (\v2.v1 (\p.p v2 ctx))))
                            (uni0 (\v2.cnt (\ctx.app (v1 ctx) (v2 ctx))))))
             (b1 (cnt (\ctx.ctx b1))
                 (\d\d.uni0 (\v.cnt (\ctx.v (ctx b1))) ps0 ys)) ps1)) uni0 cnt
  in uni0;

  -- data T = A (V -> V)  -- abstraction
  --        | V           -- free variable
  --        | O           -- "Omega" marking the end of the available input
  caseT = \t\a\v\o. t a v o;
  A = \f. \a\v\o. a f;
  V = \a\v\o. v;
  O = \a\v\o. o;

  nfT = \t. caseT t
    (\f. nfT (f V))  -- A: apply to a free variable to further reduce whnf
    t                -- V: done, this is a normal form
    t;               -- O: done, we've hit the "Omega"

  appT = \t\u. caseT t
    (\f. f u)        -- A: apply
    (nfT u)          -- V: reduce argument to normal form next
    t;               -- O: preserve O
  appT = \t. caseT t (\f. f) nfT (\u. t); -- same but smaller

  absT = A;
```

```
  boolT = \x. absT (\t. absT (\f. x t f));
  consT = \x\xs. absT (\p. appT (appT p x) xs);

  result = \pspi\rs. \p.
    let uni1 = uni id id (\prog\ps\rs. prog ps rs) id in
    p (uni1 io) (uni1 rs pspi);

  rest = \prog\ps.
    let go = \is\pspi\xs.
         caseT (nfT (appT (prog xs) (is 0)))
            xs -- A: impossible case
            (result (pspi nil) xs) -- V: we've found the remainder
            -- 0: start over with a longer input list
            (xs (\x\xs. go (\tl. is (consT (boolT x) tl))
                               (\tl. pspi (\p. p x tl)) xs))
    in  go id ps;

  pairup = uni absT appT rest id io;
in pairup
```

Although more involved, our program is almost a quarter the size of Chaitin's when measured in bits. Chaitin also offered a program of size 2104 bits, at the cost of introducing yet another primitive into his language, which is still 2.7x longer than ours.

6 Future Research

It would be nice to have an objective measure of the simplicity and expressiveness of a universal machine. Sizes of constants in fundamental theorems are an indication, but one that is all too easily abused. Perhaps diophantine equations can serve as a non-arbitrary language into which to express the computations underlying a proposed definition of algorithmic complexity, as Chaitin has demonstrated for relating the existence of infinitely many solutions to the random halting probability Ω. Speaking of Ω, our model provides a well-defined notion of halting as well, namely when $\mathbf{U}\ (p : z) = \langle M, z\rangle$ for any term M (we might as well allow M without normal form). Computing upper and lower bounds on the value of Ω_λ, as Chaitin did for his LISP-based Ω, and Calude et al. for various other languages, should be of interest as well. A big task remains in finding a good constant for the other direction of the 'Symmetry of Information' theorem, for which Chaitin has sketched a program. That constant is bigger by an order of magnitude, making its optimization an everlasting challenge.

7 Conclusion

The λ-calculus is a surprisingly versatile and concise language, in which not only standard programming constructs like bits, tests, recursion, pairs and lists, but also reflection, reification, and marshalling are readily defined, offering an elegant concrete foundation of algorithmic information theory.

An implementation of Lambda Calculus, Combinatory Logic, along with their binary and universal versions, written in Haskell, is available at [27].

8 Acknowledgements

I am greatly indebted to Paul Vitányi for fostering my research into concrete definitions of Kolmogorov complexity, and to Robert Solovay, Christopher Hendrie and Bertram Felgenhauer for illuminating discussions on my definitions and improvements in program sizes.

References

[1] R. Penrose, *The Emperor's New Mind*, Oxford University press, 1989.

[2] G. Chaitin, *An Invitation to Algorithmic Information Theory*, DMTCS'96 Proceedings, Springer Verlag, Singapore, 1997, pp. 1–23 (http://www.cs.auckland.ac.nz/CDMTCS/chaitin/inv.html).

[3] G. Chaitin, *Exploring Randomness*, Springer Verlag, 2001. (http://www.cs.auckland.ac.nz/CDMTCS/chaitin/ait3.html)

[4] R. Muller, *M-LISP: A representation-independent dialect of LISP with reduction semantics*, ACM Transactions on Programming Languages and Systems 14(4), 589–616, 1992.

[5] L. Levin, *On a Concrete Method of Assigning Complexity Measures*, Doklady Akademii nauk SSSR, vol. 18(3), pp. 727–731, 1977.

[6] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Graduate Texts in Computer Science, second edition, Springer-Verlag, New York, 1997.

[7] S.C. Kleene, *Lambda-Definability and Recursiveness*, Duke Mathematical Journal, 2, 340–353, 1936.

[8] Vladimir Levenshtein, *On the redundancy and delay of decodable coding of natural numbers*, Systems Theory Research, 20, 149–155, 1968.

[9] Frank Pfenning and Conal Elliot, *Higher-Order Abstract Syntax*, ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, 199–208, 1988.

[10] D. Friedman and M. Wand, *Reification: Reflection without Metaphysics*, Proc. ACM Symposium on LISP and Functional Programming, 348–355, 1984.

[11] Gudmund S. Frandsen and Carl Sturtivant, *What is an Efficient Implementation of the λ-calculus?*, Proc. ACM Conference on Functional Programming and Computer Architecture (J. Hughes, ed.), LNCS 523, 289–312, 1991.

[12] J. Steensgaard-Madsen, *Typed representation of objects by functions*, TOPLAS 11-1, 67–89, 1989.

[13] N.G. de Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation*, Indagationes Mathematicae 34, 381–392, 1972.

[14] H.P. Barendregt, *Discriminating coded lambda terms*, in (A. Anderson and M. Zeleny eds.) Logic, Meaning and Computation, Kluwer, 275–285, 2001.

[15] Bertram Felgenhauer, private communication, aug 27, 2011.

[16] François-Nicola Demers and Jacques Malenfant, *Reflection in logic, functional and object-oriented programming: a Short Comparative Study*, Proc. IJCAI Workshop on Reflection and Metalevel Architectures and their Applications in AI, 29–38, 1995.

[17] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes, *Essentials of Programming Languages – 2nd ed*, MIT Press, 2001.

[18] Oleg Mazonka and Daniel B. Cristofani, *A Very Short Self-Interpreter*, `http://arxiv.org/html/cs.PL/0311032`, 2003.

[19] D. Hofstadter, *Godel, Escher, Bach: an Eternal Golden Braid*, Basic Books, Inc., 1979.

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

[21] J.R. Kennaway and J. W. Klop and M.R. Sleep and F.J. de Vries, *Infinitary Lambda Calculus*, Theoretical Compututer Science, 175(1), 93–125,1997.

[22] Simon Peyton Jones, *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, in "Engineering theories of software construction", ed. Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, 47–96, 2001.

[23] The Haskell Home Page, `http://haskell.org/`.

[24] Brainfuck homepage, `http://www.muppetlabs.com/~breadbox/bf/`.

[25] Torben Æ. Mogensen, *Linear-Time Self-Interpretation of the Pure Lambda Calculus*, Higher-Order and Symbolic Computation 13(3), 217-237, 2000.

[26] D. A. Turner, *Another algorithm for bracket abstraction*, J. Symbol. Logic 44(2), 267–270, 1979.

[27] J. T. Tromp, `http://tromp.github.io/cl/cl.html`, 2004.