Computation Theory

Lecture Notes

Gert Smolka
Saarland University
January 28, 2015

1 Introduction

We study the expressivity of a minimal functional programming language L in constructive type theory.[1] On the positive side we show that L can compute with numbers and programs and can express a self-interpreter. On the negative side we show that most program properties cannot be decided with L. L is Turing-complete in that it can express an interpreter for Turing machines.

The main goal of our study is a formal and purely constructive development of a basic theory of computation. We will write actual programs and prove that they meet their specifications. This is in sharp contrast to the usual treatment of computation theory in textbooks, where the existence of complex programs like self-interpreters is claimed without mathematical proof and without giving the program.

For a formal study of computation the choice of the underlying programming language is crucial. Turing machines are a bad choice since writing and verifying Turing machines is difficult and cannot build on common practice in programming. We use a minimal functional language L where programming and verification can follow common practice.

We refer the reader to Yannick's Forsters Bachelor's thesis [4], which presents a formal and constructive development of basic computation theory carried out in Coq. The language L used in our notes is a minor variant of the language L used in Forster's development.

2 Programming Language L

We will work with a functional programming language L where all data objects (e.g., booleans, numbers, programs) are represented as functions. L is a much

[1] We do not assume excluded middle in this note.

simplified version of Church's untyped lambda calculus [1]. Technically speaking, L is a weak call-by-value lambda calculus based on capturing substitution. A variant of L is investigated by Dal Lago and Martini [6] who show that programs in L can simulate Turing machines with polynomial overhead.

Lambda calculus existed before Turing conceived his machines. Turing [9, 8] himself showed that computability in the lambda calculus agrees with computability based on Turing machines. The first undecidability result ever was obtained by Church [2] based on the lambda calculus.

The syntax of L consists of **variables** (x, y, z) and **terms** (s, t, u) defined as follows:

$$s,t,u \;::=\; x \mid \lambda x.s \mid st \qquad (x \in \mathbf{N})$$

Terms of the form $\lambda x.s$ are called **abstractions**, and terms of the form st are called **applications**. The syntax of L is familiar from functional programming and type theory and we adopt the conventions used there.

A variable x is **free** in a term s if it has an occurrence in s that is not in the scope of an abstraction $\lambda x.t$. A term is **closed** if it has no free variable. For instance, the term $\lambda xy.zxy$ has a single free variable, which is z. Here are examples of closed terms that we will use in the following.

$$I := \lambda x.x \qquad K := \lambda xy.x \qquad \omega := \lambda x.xx \qquad \Omega := \omega\omega$$

A term is **open** if it is not closed. A **procedure** is a closed abstraction, and a **program** is a closed term. The terms I, K, and ω are typical examples of procedures, and $I\omega$ and Ω are examples of programs.

A **substitution** s^z_u yields the term obtained from s by replacing every free occurrence of z with the term u. For instance, $(\lambda x.xy)^y_{yx} = \lambda x.x(yx)$. Note that we use **capturing** substitution (e.g., in the previous example the outer x is captured by the abstraction $\lambda x.x(yx)$). Capturing is not admissible when substitution is used logically. It can always be avoided by renaming local variables (i.e., $\lambda x_1.x_1(yx)$ in our example). Capturing can only occur with a substitution s^z_u where u is an open term. Capturing is not a problem for L since when it matters u will always be a closed term.

We define substitution by structural recursion on terms:

$$\begin{aligned} x^z_u &:= \text{if } x{=}z \text{ then } u \text{ else } x \\ (\lambda x.s)^z_u &:= \text{if } x{=}z \text{ then } \lambda x.s \text{ else } \lambda x.\, s^z_u \\ (st)^z_u &:= (s^z_u)(t^z_u) \end{aligned}$$

Fact 1 A term s is closed if and only if $s^z_u = s$ for every z and u.

Reduction is a binary predicate $s \succ t$ on terms describing single computation steps. We define reduction inductively using substitution.

$$\frac{}{(\lambda x.s)(\lambda y.t) \succ s^x_{\lambda y.t}} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

The first rule provides a restricted form of β-reduction, and the other two rules make it possible to descent into applications. Note that there is no descent rule for abstractions. This means that abstractions are irreducible.

Fact 2 Reduction preserves closedness.

Fact 3 A closed term is irreducible if and only if it is an abstraction.

We define the predicates $s \succ^n t$ and $s \rhd t$ as follows:

$$\frac{}{s \succ^0 s} \qquad \frac{s \succ u \qquad u \succ^n t}{s \succ^{n+1} t}$$

$$s \rhd t \;:=\; \exists n.\, s \succ^n t$$

Note that $s \succ^n t$ says that t can be obtained with n reduction steps from s, and that $s \rhd t$ is the reflexive and transitive closure of $s \succ t$.

Our definition of reduction is indeterministic in that a term may reduce to different terms is one step. For instance, $(II)(II) \succ I(II)$ and $(II)(II) \succ (II)I$. It turns out that this indeterminism neither affects termination nor the final outcome of a reduction.

Fact 4 (Diamond Property) If $s \succ t_1$ and $s \succ t_2$, then either $t_1 = t_2$ or there exists a term u such that $t_1 \succ u$ and $t_2 \succ u$.

Proof By induction on s. ∎

Fact 5 (Uniform Confluence) If $s \succ^m t_1$ and $s \succ^n t_2$, then there exist numbers k and l and a term u such that $t_1 \succ^k u$ and $t_2 \succ^l u$ and $m + k = n + l$.

Proof One first shows the claim for $n = 1$ by induction on m using the diamond property. With this special case the claim follows by induction on n. ∎

Fact 6

1. $s \rhd t$ is reflexive and transitive.
2. If $s \rhd s'$, then $st \rhd s't$.
3. If $t \rhd t'$, then $st \rhd st'$.

4. If $s \succ t$, then $s \triangleright t$.

We say that s **evaluates to** t and write $s \Downarrow t$ if $s \triangleright t$ and t is irreducible. We say that a term **terminates** if it evaluates to some term. We say that a term **diverges** if it does not terminate. The term Ω diverges since $\Omega \succ \Omega$. If $s \Downarrow t$, we say that t is the **normal form** of s.

Fact 7

1. If $s \triangleright t$, then $s \Downarrow u$ if and only if $t \Downarrow u$.
2. If $s \Downarrow t_1$ and $s \Downarrow t_2$, then $t_1 = t_2$.
3. If $s \triangleright t$, then s terminates if and only if t terminates.
4. If $s \triangleright t$, then s diverges if and only if t diverges.

Fact 8

1. If st terminates,then both s and t terminate.
2. If s or t diverges, then st diverges.

Exercise 9 Give a formal definition of free variables.

Exercise 10 Prove the facts stated in this section.

3 Booleans, Numbers, and Recursion

We can say that L computes with procedures and that all values in L are procedures. If the application of a procedure to a procedure terminates, it terminates with a procedure. Thus a procedure can be applied to as many procedures as one likes. The first such purely functional programming language was Church's untyped lambda calculus. L is a much simplified version of the untyped lambda calculus.

We represent the booleans with the following procedures:[2]

$$\overline{true} := \lambda xy.x$$
$$\overline{false} := \lambda xy.y$$

A conditional (if s then t else u) can be expressed as stu. This form of conditional is strict in that it evaluates both t and u. A usual **non-strict conditional** can be expressed as $s(\lambda x.t)(\lambda x.u)I$ where x is not free in t or u. For instance, $\overline{true}\ \overline{true}\ \Omega$ diverges while $\overline{true}\ (\lambda x.\overline{true})\ (\lambda x.\Omega)\ I$ evaluates to $\overline{true}$.

[2] When we define concrete terms, x stands for 0, y for 1, z for 2, f for 3, and g for 4. Note that with this convention the terms $\lambda x.x$ and $\lambda y.y$ are different.

We represent the naturals numbers as follows:

$$\begin{aligned} \overline{0} &:= \lambda xy.x \\ \overline{n+1} &:= \lambda xy.y\,\overline{n} \end{aligned}$$

A match ($\mathsf{match}\ s\ \mathsf{with}\ 0 \Rightarrow t \mid Sz \Rightarrow u$) can now be represented as $s\,t\,(\lambda z.u)$. This representation of a match is strict for its first argument. A non-strict match can be obtained with $s\,(\lambda x.t)\,(\lambda x.zu)\,I$ where x occurs neither in t nor u. We define the successor function as follows:

$$succ := \lambda zxy.yz$$

Fact 11 $succ\,\overline{n} \rhd \overline{n+1}$.

The procedural encoding of numbers shown above relies on a scheme devised by Scott that works for all constructor types. Scott's encoding of the numbers appears in [3, 11]. Church devised a different encoding for numbers incorporating primitive recursion (so-called Church numerals [1]).

For the definition of addition we need recursion. L can express recursion with a recursion operator ρ that given a procedure s yields a procedure s^ρ such that

$$(s^\rho)t \rhd s(s^\rho)t$$

for every procedure t. Now an addition procedure can be obtained from a non-recursive addition procedure

$$Add := \lambda fxy.\ xy(\lambda z.\ succ\ (fzy))$$

taking the recursive procedure as first argument. Given the characteristic property of ρ stated above, it is easy to verify that the procedure Add^ρ satisfies the reductions

$$\begin{aligned} Add^\rho\ \overline{0}\ \overline{n} &\rhd \overline{n} \\ Add^\rho\ \overline{m+1}\ \overline{n} &\rhd succ\ (Add^\rho\ \overline{m}\ \overline{n}) \end{aligned}$$

We define the **recursion operator** ρ as follows:

$$\begin{aligned} A &:= \lambda zg.\ g(\lambda x.zzgx) \\ s^\rho &:= \lambda x.\ AAsx \end{aligned}$$

Fact 12 $(s^\rho)t \rhd s(s^\rho)t$ if s is a procedure and t is an abstraction.

We now define a procedure for addition:

$$\begin{aligned} Add &:= \lambda fxy.\ xy(\lambda z.\ succ\ (fzy)) \\ add &:= Add^\rho \end{aligned}$$

Fact 13

1. $add\ \overline{0}\ \overline{n} \triangleright \overline{n}$
2. $add\ \overline{m+1}\ \overline{n} \triangleright succ\ (add\ \overline{m}\ \overline{n})$
3. $add\ \overline{m}\ \overline{n} \triangleright \overline{m+n}$

Proof Claim (3) follows by induction on m from claims (2) and (1). Claims (1) and (2) are easily verified. Here is the verification for Claim (2).

$$\begin{aligned} add\ \overline{m+1}\ \overline{n} &\triangleright Add\ add\ \overline{m+1}\ \overline{n} && \text{Fact 12} \\ &\triangleright \overline{m+1}\ \overline{n}\ (\lambda z.\ succ\ (add\ z\ \overline{n})) \\ &\triangleright (\lambda z.\ succ\ (add\ z\ \overline{n}))\ \overline{m} \\ &\triangleright succ\ (add\ \overline{m}\ \overline{n}) \end{aligned}$$

■

Note that the recursion operator ρ obtains recursion by means of self application. The technique is well-known from untyped lambda calculus, where recursive functions can be obtained as fixed points of the base function. A recursion operator for untyped lambda calculus was first given by Turing [10].

We now know that L can express datatypes, matches, and recursion. This is enough to write an interpreter for Turing machines. We conclude that L is Turing-complete.

Programming in L is functional programming: We represent data with constructor types and define functions that recurse over constructor types. By construction functions on constructor types satisfy basic characteristic reductions, from which by induction we can obtain the reduction expressing the correctness of the function. Fact 13 states the basic characteristic reductions and the reduction expressing correctness for the addition function *add*.

In functional programming one usually speaks of equations where we speak of reductions. Reductions can be seen as directed equations. One suitable equality relation for L would be the least equivalence relation on terms containg reduction.

Programming in L is more powerful than programming in constructive type theory in that we can use general recursion rather than structural recursion. As a result we can write functions that may not terminate for certain arguments.

Exercise 14 Let s be a closed term. Give a closed term t such that $t \triangleright st$.

Exercise 15 Write and verify a multiplication procedure.

Exercise 16 Devise a procedural representation for lists and verify procedures for cons and append.

4 Procedural Term Representation and L-Decidability

Terms are formalized as an inductive type with three constructors in type theory. This yields the following procedural representation:[3]

$$\begin{aligned} \ulcorner n \urcorner &:= \lambda xyz.\, x\ \overline{n} \\ \ulcorner \lambda n.s \urcorner &:= \lambda xyz.\, y\ \overline{n}\ \ulcorner s \urcorner \\ \ulcorner st \urcorner &:= \lambda xyz.\, z\ \ulcorner s \urcorner\ \ulcorner t \urcorner \end{aligned}$$

Exercise 17 Write procedures satisfying the following specifications.

a) $var\ \overline{n} \rhd \ulcorner n \urcorner$

b) $lam\ \overline{n}\ \ulcorner s \urcorner \rhd \ulcorner \lambda n.s \urcorner$

c) $app\ \ulcorner s \urcorner\ \ulcorner t \urcorner \rhd \ulcorner st \urcorner$

d) $Q\ \ulcorner s \urcorner \rhd \ulcorner\ulcorner s \urcorner\urcorner$

Terms are a versatile universal data structure subsuming numbers, trees, and programs. With the procedural representation of terms we can write programs that operate on programs.

We shall study predicates on terms that can be decided by procedures in L. We write T for the type of terms and consider predicates $p : \mathsf{T} \to \mathit{Prop}$. A procedure u is an **L-decider** for a predicate p if

$$\forall s.\ \ (ps \land u\ulcorner s \urcorner \rhd \overline{true}) \ \lor\ (\neg ps \land u\ulcorner s \urcorner \rhd \overline{false})$$

A predicate is **L-decidable** if it has an L-decider.

To prove that a predicate p is L-decidable we have to provide a procedure u and a proof showing that the procedure u decides p.

We now show that termination of terms is an undecidable problem.

Fact 18 If p is L-decidable, then p is propositionally decidable (i.e., $ps \lor \neg ps$ for every term s).

Fact 19 The class of decidable predicates is closed under negation, conjunction, and disjunction.

We define reducibility and termination of terms as follows:

$$\begin{aligned} \mathit{red}\ s &:= \exists t.\ s \succ t && s\ \textit{reduces} \\ \mathit{ter}\ s &:= \exists t.\ s \rhd t \land \neg \mathit{red}\ s && s\ \textit{terminates} \end{aligned}$$

Lemma 20 $\lambda s.\ \mathit{ter}\ (s\ulcorner s \urcorner)$ is not L-decidable.

[3] A procedural term representation using Scott's encoding was used first by Mogensen [7].

Proof Let u be an L-decider for $\lambda s.\, ter\,(s\ulcorner s\urcorner)$. Let $v := \lambda x.\, ux(\lambda x.\Omega)II$. We consider $u\ulcorner v\urcorner$.

1. $ter\,(v\ulcorner v\urcorner)$ and $u\ulcorner v\urcorner \rhd \overline{true}$. We have $v\ulcorner v\urcorner \rhd u\ulcorner v\urcorner(\lambda x.\Omega)II \rhd \overline{true}\,(\lambda x.\Omega)II \rhd (\lambda x.\Omega)I \rhd \Omega$. Contradiction since Ω diverges.
2. $\neg ter\,(v\ulcorner v\urcorner)$ and $u\ulcorner v\urcorner \rhd \overline{false}$. We have $v\ulcorner v\urcorner \rhd u\ulcorner v\urcorner(\lambda x.\Omega)II \rhd \overline{false}\,(\lambda x.\Omega)II \rhd II \rhd I$. Contradiction since I terminates. ∎

Theorem 21 Termination of terms is not L-decidable.

Proof Follows from Lemma 20. ∎

Exercise 22 Show that termination of procedures is not L-decidable. Hint: Prove Lemma 20 for procedures.

5 Scott's Theorem

We now show that nontrivial extensional program properties are not L-decidable. This result is known as Rice's Theorem in conventional computation theory [5] and as Scott's Theorem in the literature on untyped lambda calculus [1]. Scott's theorem rests on a fact that it sometimes referred to as *second fixed point theorem*.

Fact 23 For every program s there exists a program t such that $t \rhd s\ulcorner t\urcorner$.

Proof We use the procedures from Exercise 17. Let $A := \lambda x.\, s\,(app\; x\,(Qx))$ and $t := A\ulcorner A\urcorner$. We have $t \rhd s\,(app\ \ulcorner A\urcorner\,(Q\ulcorner A\urcorner)) \rhd s\ulcorner A\ulcorner A\urcorner\urcorner = s\ulcorner t\urcorner$. ∎

Theorem 24 (Scott) No nontrivial extensional class of programs is L-decidable.

Proof Let p be a predicate $\mathbf{T} \to Prop$ as follows:

1. If s is a program such that $s \rhd t$, then ps if and only if pt.
2. There are programms t_1 and t_2 such that pt_1 and $\neg pt_2$.
3. u is an L-decider for p.

We show that the assumptions are contradictory. Let

$$v := \lambda x.\, ux(\lambda x.t_2)(\lambda x.t_1)I$$

By Fact 23 we have a program t such that $t \rhd v\ulcorner t\urcorner$. Thus $t \rhd u\ulcorner t\urcorner(\lambda x.t_2)(\lambda x.t_1)I$. Case analysis on the fact that u L-decides p.

1. pt and $u\ulcorner t\urcorner \rhd \overline{true}$. Then $t \rhd t_2$. Contradiction since $\neg pt_2$.

2. $\neg pt$ and $u\ulcorner t\urcorner \rhd \overline{false}$. Then $t \rhd t_1$. Contradiction since pt_1. ∎

Exercise 25 Let s and t be procedures. Show that the class of programs u such that $us \rhd t$ is not L-decidable.

Exercise 26 Show that there exists a program t that computes its procedural representation $\ulcorner t \urcorner$, that is, $t \rhd \ulcorner t \urcorner$. Hint: Use Fact 23.

6 Step-Indexed Evaluation

We have defined an evaluation predicate $s \Downarrow t$ for L and shown that it is functional. Since there are nonterminating programs, the evaluation predicate is not total. Given that the evaluation predicate corresponds to a Turing-complete programming system, we cannot expect that it can be captured in constructive type theory with a total function $\mathbf{T} \to \mathbf{T}_\perp$ where $\mathbf{T}_\perp$ is the option type over $\mathbf{T}$ (recall that $\mathbf{T}$ is the type of terms). However, if we add a resource bound as additional argument, we can capture evaluation in L with a function

$$eva \ : \ \mathbf{N} \to \mathbf{T} \to \mathbf{T}_\perp$$

satisfying the properties stated in Fact 27. We will call a function a **step-indexed interpreter** if it satisfies the specification for *eva*. Step-indexed interpreters may differ in how they use the bound given as argument. We will use a step-indexed interpreter where the bound is taken as a bound on the recursion depth.

Here is the definition of the function *eva* we will use. Note that *eva* is defined by structural recursion on its first argument n.

$$\begin{aligned}
&eva : \mathbf{N} \to \mathbf{T} \to \mathbf{T}_\perp \\
&\quad eva\ n\ x = \perp \\
&eva\ n\ (\lambda x.s) = \lfloor \lambda x.s \rfloor \\
&\quad eva\ 0\ (st) = \perp \\
&eva\ (Sn)\ (st) = \mathsf{match}\ eva\ n\ s,\ eva\ n\ t\ \mathsf{with} \\
&\qquad\qquad \mid \lfloor \lambda x.s \rfloor,\ \lfloor t \rfloor \Rightarrow eva\ n\ s_t^x \\
&\qquad\qquad \mid _\ _ \Rightarrow \perp
\end{aligned}$$

Fact 27 Let s be a program. Then:

1. $s \Downarrow t$ if and only if $eva\ n\ s = \lfloor t \rfloor$ for some n.
2. If $eva\ n\ s = \lfloor t \rfloor$, then $eva\ (Sn)\ s = \lfloor t \rfloor$.

7 L-Decidability versus C-Decidability

We call a predicate $p : \mathbf{T} \to \mathit{Prop}$ **C-decidable** if there exists a boolean function $f : \mathbf{T} \to \mathbf{B}$ for which we can prove $\forall s.\ ps \leftrightarrow fs = \mathit{true}$.

Theorem 28 Every L-decidable predicate is C-decidable.

Proof Let u be an L-decider for p. By Fact 27 we know that for every term s there exists a number n such that

$$\mathit{eva}\ n\ (u\ulcorner s\urcorner) = \lfloor\overline{\mathit{true}}\rfloor \ \vee\ \mathit{eva}\ n\ (u\ulcorner s\urcorner) = \lfloor\overline{\mathit{false}}\rfloor$$

Since equality on $\mathbf{T}_\perp$ is decidable, constructive choice for $\mathbf{N}$ gives us a function σ such that

$$\mathit{eva}\ (\sigma s)\ (u\ulcorner s\urcorner) = \lfloor\overline{\mathit{true}}\rfloor \ \vee\ \mathit{eva}\ (\sigma s)\ (u\ulcorner s\urcorner) = \lfloor\overline{\mathit{false}}\rfloor$$

for every term s. We now obtain a boolean decision function f for p as follows:

$$\begin{aligned} f s \ =\ & \mathsf{match}\ \mathit{eva}\ (\sigma s)\ (u\ulcorner s\urcorner)\ \mathsf{with} \\ & \mid \lfloor\overline{\mathit{true}}\rfloor \ \Rightarrow\ \mathit{true} \\ & \mid \lfloor\overline{\mathit{false}}\rfloor \ \Rightarrow\ \mathit{false} \\ & \mid \perp \ \Rightarrow\ \mathit{false} \end{aligned}$$

■

At first, Theorem 28 seems to be in conflict with conventional computation theory since it seems to say that every predicate decidable in a Turing-complete system is decidable in the terminating system of Coq's constructive type theory. The conflict is resolved by the observation that the theorem assumes a correctness proof for the L-decider. Thus the mathematical result that a terminating programming system always misses total functions expressible in a Turing-complete system implies that there are total Turing-computable functions whose correctness cannot be shown in a consistent extension of constructive type theory.

Fact 29 Together, the following assumptions are inconsistent.

1. Every predicate $\mathbf{T} \to \mathit{Prop}$ is C-decidable.
2. Every C-decidable predicate is L-decidable.

Proof Follows from the fact that there are predicates that are not L-decidable (Lemma 20). ■

It is common belief that Coq's type theory is consistent with the assumption that every predicate is C-decidable. A proof in Coq that C-decidability implies L-decidability would falsify this belief.

Exercise 30 Show that excluded middle follows from the assumtion that every predicate $\mathbf{T} \to \mathit{Prop}$ is C-decidable.

8 Self-Interpretation

The step-indexed interpreter for L (a function in type theory) can be internalized as a procedure in L. This way we can obtain a self-interpreter for L which gives us the equivalent of a universal Turing machine.

Theorem 31

1. There is a procedure u such that $u\ \overline{n}\ \ulcorner s \urcorner \rhd \ulcorner eva\ n\ s \urcorner$ for every number n and every term s. The procedural representation for elements of $\mathbf{T}_\bot$ is defined as follows: $\ulcorner \lfloor s \rfloor \urcorner := \lambda xy.x\ulcorner s \urcorner$ and $\ulcorner \bot \urcorner := \lambda xy.y$.
2. There is a procedure u such that for all programs s and t the following holds:
   a) $u\ulcorner s \urcorner \rhd \ulcorner t \urcorner$ if and only if $s \Downarrow t$.
   b) $u\ulcorner s \urcorner$ terminates if and only if s terminates.
3. There is a procedure u such that for all programs s and t the following holds:
   a) If s or t terminates, then $u\ulcorner s \urcorner\ulcorner t \urcorner$ terminates.
   b) If $u\ulcorner s \urcorner\ulcorner t \urcorner$ terminates, then either $u\ulcorner s \urcorner\ulcorner t \urcorner \rhd \overline{true}$ and s terminates or $u\ulcorner s \urcorner\ulcorner t \urcorner \rhd \overline{false}$ and t terminates.

References

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

[2] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[3] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. North-Holland Publishing Company, 1972.

[4] Yannick Forster. A formal and constructive theory of computation. Bachelor's thesis, Saarland University, `www.ps.uni-saarland.de/~forster/bachelor.php`, 2014.

[5] Dexter C. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer-Verlag, 1997.

[6] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[7] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

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

[9] Alan M. Turing. The þ-function in λ-K-conversion. *The Journal of Symbolic Logic*, 2(04):164–164, 1937.

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

[11] Christopher Wadsworth. Some unusual λ-calculus numeral systems. In Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 215–230. Academic Press, 1980.