A Swiss Pocket Knife for Computability

Neil D. Jones
Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
`neil@diku.dk`

This research is about operational- and complexity-oriented aspects of classical foundations of computability theory. The approach is to re-examine some classical theorems and constructions, but with new criteria for success that are natural from a programming language perspective.

Three cornerstones of computability theory are the *S-m-n* theorem; Turing's "universal machine"; and Kleene's second recursion theorem. In today's programming language parlance these are respectively partial evaluation, self-interpretation, and reflection. In retrospect it is fascinating that Kleene's 1938 proof is constructive; and in essence builds a self-reproducing program.

Computability theory originated in the 1930s, long before the invention of computers and programs. Its emphasis was on delimiting the boundaries of computability. Some milestones include 1936 (Turing), 1938 (Kleene), 1967 (isomorphism of programming languages), 1985 (partial evaluation), 1989 (theory implementation), 1993 (efficient self-interpretation) and 2006 (term register machines).

The "Swiss pocket knife" of the title is a programming language that allows efficient computer implementation of all three computability cornerstones, emphasising the third: Kleene's second recursion theorem. We describe experiments with a tree-based computational model aiming for both fast program generation and fast execution of the generated programs.

1 Introduction

1.1 Context

"The grand confluence" of the 1930s (term due to Gandy [7]) was a first major accomplishment of computability theory: the realisation that the classes of computable problems defined using Turing machines, lambda calculus, register machines, recursion schemes, rewrite systems,..., *are all identical*. This led to a deep mathematical theory (recursive function theory) about the boundary between computable and uncomputable problems, e.g., by Kleene, Turing, McCarthy, Rogers [18, 17, 21, 22, 27, 30]. Three cornerstones of computability theory were identified: the *S-m-n* theorem; the "universal machine" (as Turing called it); and Kleene's second recursion theorem.

What interests computer scientists, though, and what recursive function theory does **not** account for, is the *time it takes to compute* a function: the *size and efficiency* of the programs involved.

1.2 Contribution of this paper

Our research program is to re-examine classical computability theory constructions from an efficiency viewpoint. Some accomplishments so far: *partial evaluation* has applied one computability cornerstone, the *S-m-n theorem*, to program optimisation. The field of partial evaluation field is now substantial, e.g., as documented by Jones, Gomard, Sestoft in [15]. This paper focuses on another computability cornerstone: *Kleene's second recursion theorem* [17].

To study complexity aspects of the classical recursion-theoretic results one needs good models of computation that take into account programs' running times, as well as their expressivity. A stimulus for the current work was the elegant construction by Moss in [26], using the novel `1#` language to prove Kleene's second recursion theorem.

This paper describes computer describes constructions and experiments with a tree-based computational model aiming for both fast program generation and fast execution of the generated programs. A programming language perspective has led to a better problem understanding, and improved asymptotic efficiency. The net effect is that all three computability cornerstones can be efficiently implemented.

Advances in the first two made by a Copenhagen group and others include *constant-overhead self-interpretation*, documented in [15, 12] and other places. Efficiency improvements over [9, 26, 12] include a *constant-time implementation* of Kleene's second recursion theorem as by Bonfante and Greenbaum in [2], and constant-time versions of the constructions by Moss in [26].

2 Fundamentals

2.1 Recursive function theory

Rogers' axioms: an "acceptable" programming language consists of[1]

- Two sets, *Pgms* and D (of programs $p,q,e,\ldots$ and data $d,s,x,y,\ldots$), with $Pgms \subseteq D$.
- A *semantic function*

  $$[\![p]\!]^n : D^n \rightharpoonup D$$

  associating with each program and each n a partial n-ary function. (We omit n in $[\![p]\!]^n$ when it is obvious from context.) The semantic function must have these properties:

  1. *Turing completeness*: a (mathematical partial) function $f : D^n \rightharpoonup D$ is computable *if and only if* $f = [\![p]\!]^n$ for some program p.
  2. *Universal program property*[2]:

     $$\exists univ \in Pgms\ \forall p \in Pgms\ \forall d \in D\ (\ [\![p]\!](d) = [\![univ]\!](p,d)\ )$$

  3. *S-m-n property (here for $m = n = 1$)*:

     $$\exists s^1_1 \in Pgms\ \forall p \in Pgms\ \forall s,d \in D\ (\ [\![p]\!](s,d) = [\![[\![s^1_1]\!](p,s)]\!](d)\ )$$

Restated in computer science terms: a universal machine *univ* is a *self-interpreter*; and an *S-1-1* program s^1_1 is a *partial evaluator* or *program specialiser*. (Remark: Rogers has proven in [27] the remarkable result that any two acceptable enumerations are computably isomorphic.)

[1]We follow the line of Rogers' definition of an *acceptable enumeration of the partial recursive functions* [27, 19]. Our variant: we write $[\![p]\!]$ instead of ϕ_p; and we use programs and data from the two sets Pgm, D instead of the natural numbers $\mathbb{N}$.

[2] In equations involving program semantics $=$ stands for equality of partial values, so $e_1 = e_2$ means that either e_1 and e_2 evaluate to the same value, or both are undefined (in practice, meaning: nonterminating).

2.2 Kleene's second recursion theorem

Kleene's second recursion theorem (SRT for short) is an early and very general consequence of the Rogers axioms for computability. It clearly has a flavor of *self-application*, as it in effect asserts the existence of programs that can refer to their own texts. The statement and proof are short, though the theorem's implications are many.

Theorem (Kleene, [17]) For any acceptable programming language,

$$\forall p \in Pgms\ \exists p^* \in Pgms\ \forall d \in D\ .\ [\![p]\!](p^*,d) = [\![p^*]\!](d).$$

In effect this is a *program transformation*: it takes a 2-argument program p, and constructs from it a 1-argument program p^*. Program p^* is sometimes called a "fixpoint", though it is not necessarily minimal, nor unique for the given p.[3]

The theorem can be interpreted operationally, but was proven long before the first computers were built. Applications of Kleene's theorem are many, and include:

1. A *self-reproducing* program: Suppose $[\![p]\!](q,d) = q$ for any q. Then for any input d,

   $$[\![p^*]\!](d) = [\![p]\!](p^*,d) = p^*$$

   Program p^*, when run, will print its own text, regardless of what its input was.

2. A *self-recognising* program: consider the program[4]

   $$p = \boxed{\texttt{p(q,d) = if d=q then 1 else 0}}$$

   Then $[\![p]\!](p^*,d) = [\![p^*]\!](d)$ implies: $\quad \forall d \in D \quad \begin{array}{lcl} [\![p^*]\!](d) & = & 1 \text{ if } d = p^* \\ [\![p^*]\!](d) & = & 0 \text{ if } d \neq p^* \end{array}$

   Program p^*, when run, will print 1 if its input is its own text, and print 0 otherwise.

3. Further, Kleene's theorem has many important applications in recursion theory, e.g., see the extensive overview by Moschovakis [25]. Many applications (perhaps most) require using the universal machine (a self-interpreter).

4. A major consequence of Kleene's theorem (and the reason for the theorem's name) is that it implies that *any programming language satisfying Rogers' axioms* is *closed under recursion*.[5] This is not immediately evident for, say, Turing machines since the Turing machine architecture has no recursive language constructions whatever.

   Closure of Turing-machines under recursion follows from Kleene's theorem, since this computation model satisfies Rogers' axioms. The trickiest part is the universal machine property. It was shown by Turing in [30], using a nonrecursive construction.

[3]Rogers [27] has an alternative version: that any computable total function f from programs to programs has what could be called a "syntactic fixpoint": a program p such that $[\![p]\!] = [\![f(p)]\!]$. It appears that Kleene's original version is in some sense more general than Rogers', cf. [4], so we only consider the Kleene version.

[4]We enclose program texts in boxes and use `teletype` font. Reasons: to emphasise their syntactic nature, e.g., to distinguish a program from the mathematical function it computes. In this paper programs are always imperative or first-order functional.

[5]Meaning: any function defined by recursion from programmable functions is itself programmable. See Kleene's Section 66, and the discussion around Theorems XXVI and XXVII [18].

5. *Recursion removal*: For an example, consider the program

$p =$
```
p(q,d) = if d=0 then 1 else d * univ(q, d-1)
univ (program, data) = ...  ; universal program
```

   It is immediate that

$$\begin{array}{lclcl} [\![p^*]\!](0) & = & [\![p]\!](p^*,0) & = & 1 \\ [\![p^*]\!](d+1) & = & [\![p]\!](p^*,d+1) & = & (d+1) * [\![p^*]\!](d) \end{array}$$

   Thus $[\![p^*]\!](d) = d!$, the factorial function. Net effect: p^* computes $d!$, even though *p can be defined completely without recursion* (e.g., `univ` could be Turing's universal program).

6. A mind-boggling application involves *interchanging the role of programs and data.* Consider the program

$p =$
```
p(q,d) = univ(d,q)
univ (program, data) = ...  ; universal program
```

   Then the p^* that exists by Kleene's theorem satisfies:

$$\forall\, q \in Pgms\ .\ [\![p^*]\!](q) = [\![p]\!](p^*,q) = [\![q]\!](p^*)$$

2.3 Kleene's proof of SRT

The first step, given p, is to find a program $\tilde{p}$ such that

$$\forall q \in Pgms\ \forall d \in D\ .\ [\![\tilde{p}]\!](q,d) = [\![p]\!]([\![s^1_1]\!](q,q),d)$$

Function $f(q,d) = [\![p]\!]([\![s^1_1]\!](q,q),d)$ is computable since p and s^1_1 are assumed computable. By Turing completeness, there exists a program $\tilde{p}$ to compute f.

Second step: build $p^* := [\![s^1_1]\!](\tilde{p},\tilde{p})$. Now to show that the constructed program works correctly:

$$[\![p^*]\!](d) = [\![[\![s^1_1]\!](\tilde{p},\tilde{p})]\!](d) = [\![\tilde{p}]\!](\tilde{p},d) = [\![p]\!]([\![s^1_1]\!](\tilde{p},\tilde{p}),d) = [\![p]\!](p^*,d)$$

End proof.

2.4 The Moss proof of SRT

Lawrence Moss [26] proved SRT (for a specific language `1#`) by reasoning similar to Kleene's, but with 2 computation stages and without the *S-1-1* property. First the reasoning; the `1#`-specific details are deferred to Section 3.5.

Moss' proof of SRT: The first step, given p, is to construct a program $\hat{q}$ such that

$$\forall r \in Pgms\ \forall d \in D\ .\ [\![[\![\hat{q}]\!](r)]\!](d) = [\![p]\!]([\![r]\!](r),d)$$

Section 3.5 shows that $\hat{q}$ exists in language `1#`. (Turing completeness is not enough since $[\![\hat{q}]\!]$ is nested.)

Second step: construct $p^* := [\![\hat{q}]\!](\hat{q})$. Now we show that the constructed program works correctly:

$$[\![p^*]\!](d) = [\![[\![\hat{q}]\!](\hat{q})]\!](d) = [\![p]\!]([\![\hat{q}]\!](\hat{q}),d) = [\![p]\!](p^*,d)$$

End proof.

2.5 Remarks on constructing the "fixpoint"

In both constructions only $[\![\_]\!]^1$ and $[\![\_]\!]^2$ are used: programs are run with either 1 or 2 arguments. The only connection between $[\![\_]\!]^1$ and $[\![\_]\!]^2$ is the *S-1-1* function (Kleene), or the construction of $\hat{q}$ (Moss). The programs involved in Kleene's proof have *no recursion* and *no nested loops*. (This is no surprise, since computers and programming languages had not yet been invented when the SRT was proven.)

Finally, the SRT proofs make *no use of the universal function* at all. (It is only used in applications, e.g., as in the applications above to show closure under recursion or to interchange data and program.)

3 Towards computer realisations of the SRT proof

The classical Gödel number-based constructions well-known from recursive function theory (Kleene, Rogers, Gandy[18, 27, 7]) are quite impractical to implement, as the techniques are based on numerical encoding: prime power exponentiations and factoring. Numerical encoding was reasonable for their purpose, which was to delineate the boundaries of computability and not to study complexity. We wish, however, to see how to implement such constructions efficiently on a computer.

A critical step in Kleene's proof is going from the *mathematical function* f to the *program* $\tilde{p}$. By appealing to Turing completeness, the Kleene proof avoids being tied to any one programming language. Our goals are different, and to talk about SRT complexity, we will need both *more concrete computation models* and *explicit program constructions*.

3.1 The "Swiss pocket knife"

The imperative flow chart language of [15] is enough to carry out all the SRT applications above. We will see, as did Bonfante and Greenbaum in [2], that *the constructions to prove the SRT* can be done in a very small subset TINY of the flow chart language of [15] such that *all programs run in constant time*.[6]

3.2 Programs as data

Programs have been formulated in computability theory in many ways, e.g., as a natural number (a Gödel number, by Kleene and others [18, 17, 27]); a Turing machine program or set of quintuples [30]; a lambda expression by Church [5]; a set of recursive function definitions (by Kleene and others [18]); a set of rewrite rules or a register machine (see Minsky in [22]); an S-expression in McCarthy's LISP [21]), and many others. In classical recursive function theory a program is a natural number from $\mathbb{N}$.

A programming language view is that a program is *an abstract syntax tree* (e.g., an S-expression as in SCHEME or LISP). Advantages: abstract syntax trees such as S-expressions give more natural versions of the *size* $|p|$ or $|d|$ of a program p or a data value d. This is important because the relation between input size and program running time is central in computer science, cf. the P=NP problem.

More accurately, the semantics of LISP-like languages are not really based on trees, but rather on *DAGs* (directed acyclic graphs), since substructures of data may be (and usually are) shared. This can be critical for measuring running times.[7]

[6]The TINY language is of course not Turing-complete. In brief, the Turing-complete imperative languages of [12, 13, 15] are in essence TINY plus `while` and `if` commands.

[7]Tree-based and DAG-based models define the same input-output relations for programs without selective updating such as `set-car!`.

Amtoft et al. [9] tried Kleene's SRT construction in a first-order LISP-like functional programming language with tree-structured data, encountered problems, and modified the language. See Section 3.4.

3.3 The Kleene SRT proof with tree-structured data

To make Kleene's construction computationally explicit one can use imperative flow chart programs with LISP-like data as in [15] Chapter 4. A very small subset suffices for this paper: the TINY language of Bonfante and Greenbaum [2]. Program format: $p = \boxed{\texttt{read x1,...,xn; C; write out}}$ with $n \geq 0$. Here `C` is a command built from assignments `x := e`, sequencing `C1;C2` and expressions with variables, constants `'d`, and operators `hd`, `tl`, `cons`. There are no tests or loops, so *every program will run in constant time* (assuming as usual in DAG-based semantics that operators `hd`, `tl`, `cons` are constant-time).

Program specialisation: Let program $p = \boxed{\texttt{read q, d; C; write out}}$, and let s be a "static" value for variable `q`. The result of specialisation could be

$$p' = \boxed{\texttt{read d; q := 's; C; write out}}$$

This specialisation gets p' by removing from p the input of variable `q`, and adding assignment `q := 's`. It inserts the static data value `s` *inside* the constant `'s`. More generally, we need a concrete program s_1^1 such that

$$\forall pgm\, \forall s,d\ .\ [\![pgm]\!](s,d) = [\![[\![s_1^1]\!](pgm,s)]\!](d)$$

The form of the specialiser is

$$s_1^1 = \boxed{\texttt{read pgm, s;}\ \ \texttt{C}_{spec}\texttt{;}\ \ \texttt{write outpgm}}$$

where command $\texttt{C}_{spec}$ is the "body" of s_1^1. To proceed further we need to be more specific about the form of programs as data: concrete tree structures to represent $p, p'$and s_1^1. Following the lines of [12, 13, 2] we can use a LISP-inspired *concrete syntax*, e.g.,

$$p = ((\texttt{q}\ \texttt{d})\ \ \underline{\texttt{C}}\ \ \texttt{out})$$

where $\underline{\texttt{C}}$ is a "Cambridge Polish" representation of `C`.[8] Representation of specialisation result p' above:

$$p' = ((\texttt{d})\ \ (;\ (:=\ \texttt{q}\ (\texttt{QUOTE}\ \texttt{s}))\ \ \underline{\texttt{C}})\ \ \texttt{out})$$

Obtain $[\![s_1^1]\!](p,s) = p'$ by defining the body $\texttt{C}_{spec}$ of s_1^1 to be (in LISP-like informal syntax[9]):

$C_{spec} =$
```
inputvar    := hd hd pgm; C := hd tl pgm; outputvar := hd tl tl pgm;
initialise  := list(':=, inputvar, list('QUOTE, s) );
body        := list('; ,initialise, C );
outpgm      := list(tl hd pgm, body, outputvar);
```

[8] $\underline{\texttt{'d}}$ = `(QUOTE d)`, $\underline{\texttt{op e}}$ = (`op` $\underline{\texttt{e}}$), $\underline{\texttt{cons(e,e')}}$ = (`cons` $\underline{\texttt{e}}$ $\underline{\texttt{e'}}$), $\underline{\texttt{x := e}}$ = (`:= x` $\underline{\texttt{e}}$), and $\underline{\texttt{C1;C2}}$ = (`;` $\underline{\texttt{C1}}$ $\underline{\texttt{C2}}$).

[9] More notation: `list(e1, e2,..., en)` is short for `cons(e1, cons(e2,..., cons(en, '())...))`.

3.3.1 Program details for the Kleene construction with tree-structured data

1. Let $p = \boxed{\texttt{read q,d; C}_p\texttt{; write out}}$ and let `s` be the known value for `q`. Let $\texttt{C}_{spec}$ be the body of specialiser s_1^1. As above $[\![s_1^1]\!](p,s) = p' = \boxed{\texttt{read d; q := 's; C}_p\texttt{; write out}}$.

2. Build $\tilde{p} =$

```
read q,d;
pgm := q; s := q; C_spec;  (* specialise q to q *)
q  := outpgm; C_p;         (* then run p on the result *)
write out
```

   This clearly satisfies $[\![\tilde{p}]\!](q,x) = [\![p]\!]([\![s_1^1]\!](q,q),x)$.

3. Let $p^* = [\![s_1^1]\!](\tilde{p},\tilde{p}) =$

```
read d;
q := 'p̃;                   (* initialise q to program p̃ *)
pgm := q; s := q; C_spec; (* specialise p̃ to p̃ *)
q  := outpgm; C_p;        (* run p on the result *)
write out
```

   This satisfies $[\![p^*]\!](d) = [\![[\![s_1^1]\!](\tilde{p},\tilde{p})]\!](d) = [\![\tilde{p}]\!](\tilde{p},d) = [\![p]\!]([\![s_1^1]\!](\tilde{p},\tilde{p}),d) = [\![p]\!](p^*,d)$.

Program self-reproduction: The start $\boxed{\texttt{q := '}\tilde{p}\texttt{; pgm := q; s := q; C}_{spec}\texttt{; q := outpgm}}$ of the p^* program assigns to `q` the value $[\![s_1^1]\!](\tilde{p},\tilde{p})$, which equals p^*. The net effect is that this code segment in p^* assigns to `q` *the text of the entire program p^* that contains it.*

3.3.2 Constant time, and the role of shared data-structures

It may be surprising that Kleene's SRT can be proven by such simple means. TINY is a very limited language, since a TINY program can only access (by means of `hd`, `tl`) parts of its input that lie a fixed distance from the root of its input; the program is indifferent to the remainder of its input. This implies that every TINY program runs in constant time.

An analysis of the size of $p^* = [\![s_1^1]\!](\tilde{p},\tilde{p})$: program p^* contains

- a copy of the body $\texttt{C}_p$ of program p, and a copy of the body $\texttt{C}_{spec}$ of the specialiser; plus
- a copy of program $\tilde{p}$ (in $\texttt{'}\tilde{p} = \texttt{(QUOTE }\tilde{p}\texttt{)}$). This $\tilde{p}$ *also contains copies* of both $\texttt{C}_p$ and $\texttt{C}_{spec}$.

These copies are shared in the natural implementaion: variables `pgm` and `s` in programs $\tilde{p}$ and p^* all refer to the same DAG node. One effect is that a printed-out version of p^* may be considerably larger than p^* as a DAG, beause of the shared substructures.

3.4 A reflective extension of the programming language

Amtoft et al [9] observed a practical problem in the Kleene construction in the case that program p calls the universal program *univ*. Implementing recursion as in Application 5 gave a surprise: in order to compute $n!$ the self-interpreter is applied *to interpret itself* at n meta-levels. *Consequence*: when applied to compute $n!$ the Kleene construction *takes exponential time*, and not linear time as one might expect. (Remark: the Moss construction would have the same problem.)

A design change: the functional language of [9] was given a “reflective extension”. First, a new constant `*` was added to the language. Its value: *the text of the program currently being executed*. Second, a new call form `univ p d` was added, yielding value $[\![\mathtt{p}]\!](\mathtt{d})$. With the aid of these new constructions it was straightforward to construct a program p^* as needed for the Kleene result, and without self-application. The resulting p^* evaluated $n!$ in linear time, albeit with a significant interpretation overhead.

The rationale behind this perhaps unexpected language design was that an interpreter was being used to execute programs. Since the interpreter always has to have the program it is interpreting at hand, the value of constant `*` is always available. Further, a source program call `univ p d` can be implemented by a simple recursive call to the currently running interpreter (thus sidestepping the need to interpret an interpreter, etc.).

Conclusions: the construction of [9] gives a more efficient output program than Kleene’s version; but the “reflective extension” is somewhat inelegant (even hacky); and an efficiency drawback is that *every program execution* involves a significant interpretation overhead.

3.5 The Moss SRT proof with `1#`

The Moss approach to construct $\hat{q}$ from Section 2.4 recapitulated: The language `1#` is based on *term register machines* (TRM for short): a variant of Shepherdson and Sturgis’ well-known register machines [22, 28], generalised to work on strings from $D = \{1,\#\}^*$ as data instead of natural numbers. A program operates on a fixed number of registers $R1, R2, \ldots, Rk$. To compute $[\![p]\!]^n$, the program inputs are in Registers $R1, R2, \ldots, Rn$, and output is in Register $R1$. Language `1#` is acceptable since it is possible to construct a universal program and programs for the *S-1-1* functions.

A `1#` program, as well as the data it operates on, is a string from $\{1,\#\}^*$. For program representation details, see [26]. A key point is that there exists a *program composition operation* $|$ for `1#` such that

$$\forall p,q \in Pgms\ \forall x \in D\ .\ [\![\ p \mid q\ ]\!](x) = [\![q]\!]([\![p]\!](x))$$

Operation $|$ is just “append”, i.e., string concatenation. Further, there exist terminating programs $\mathtt{move}_{i,j}$ and `write`, `diag` as follows. Their `1#` codes are also in [26], all using 3 or fewer registers.

1. $\mathtt{move}_{i,j}$ appends the contents of Ri to the right end of Rj (and empties Ri in the process).
2. For any $x \in D$, $[\![\ [\![\mathtt{write}]\!](x)\ ]\!]() = x$.
3. For any $r \in Pgm$, $[\![\ [\![\mathtt{diag}]\!](r)\ ]\!]() = [\![r]\!](r)$.

Effects: program `write` produces from input string x a program that, when run, writes x. Program `diag` produces from input r a program that, when run, computes $[\![r]\!](r)$. Conceptually, `write` expresses the essence of *code generation*; and `diag` expresses the essence of *self-application*.[10]

The Moss construction explicitly builds a program satisfying the requirements of Kleene’s proof. The first step in in the Moss SRT construction: given p, construct

$$\hat{\mathtt{q}} = \mathtt{diag} \mid \mathtt{move}_{1,2} \mid [\![\mathtt{write}]\!](\mathtt{move}_{1,4}) \mid \mathtt{move}_{2,1} \mid [\![\mathtt{write}]\!](\mathtt{move}_{4,2}) \mid [\![\mathtt{write}]\!](\mathtt{p})$$

Program $\hat{\mathtt{q}}$ is terminating since all of its parts are terminating. Given the properties of program composition and the `write`, `diag` and the $\mathtt{move}_{i,j}$ programs, it is easy to see that

$$[\![\hat{q}]\!](r) = \mathtt{move}_{1,4} \mid [\![\mathtt{diag}]\!](r) \mid \mathtt{move}_{4,2} \mid \mathtt{p}$$

which implies $[\![[\![\hat{\mathtt{q}}]\!](r)]\!](d) = [\![p]\!]([\![r]\!](r), d)$ as required. The second step is to set $p^* = [\![\hat{q}]\!](\hat{q})$.

[10] An example of a self-reproducing program similar to Application 1 is easy to construct directly: Define $\mathtt{self} := [\![\mathtt{diag}]\!](\mathtt{diag})$. Then $[\![\mathtt{self}]\!]() = [\![\ [\![\mathtt{diag}]\!](\mathtt{diag})\ ]\!]() = [\![\mathtt{diag}]\!](\mathtt{diag}) = \mathtt{self}$.

4 Operational questions about theoretical constructions

4.1 Program running times

Write $time_p(d)$ for the number of steps to compute $[\![p]\!](d)$ in a suitable computation model, e.g., a programming language. Assume given a function $time_p(d)$ that satisfies the Blum *machine-independent complexity axioms* [1, 19]:

1. For any $p \in Pgms, d \in D$, $time_p(d)$ terminates iff $[\![p]\!](d)$ terminates.
2. The property $time_p(d) \leq t$ is decidable, given program p, input data d and time t.

For instance in an imperative language one could count 1 for each executed assignment $:=$, operator, and variable or constant access.

4.2 Some natural questions for a computer scientist

1. What is the *computational overhead* of self-interpretation, i.e., applying a universal machine ?
2. Can specialisation as in the *S-1-1* axiom *speed a program up*? If so, by how much?
3. *How hard to construct* are the programs that exist by Kleene's second recursion theorem; and *how efficient* are they (or can they be)?

Questions 1 and 2 were motivated in 1971 by Futamura (reprinted in 1999 [6]); some answers are given by Jones, Gomard, Sestoft in [15]. For context, first a brief review 1 and 2 from the viewpoint of [15, 6]. Following this, we obtain some new results about question 3 (investigated earlier in [9, 26]).

4.3 Interpretation overhead

By the Rogers axiom, $[\![p]\!](d)$ terminates iff $[\![univ]\!](p,d)$ terminates. By the first Blum complexity axiom

$$\forall\ p\ \forall d\ (\exists t\ .\ time_p(d) \leq t) \text{ iff } (\exists t'\ .\ time_{univ}(p,d) \leq t')$$

Interpretation overhead is the efficiency slowdown caused by use of a universal machine, i.e., the relation between (the smallest such) t and t'. Their existence does not, however, imply there is any simple relation between them. Some possibilities for interpretation overhead:

1. $\forall p\ \forall d\ \exists c\ .\ time_{univ}(p,d) \leq c \cdot time_p(d)$ (always true)
2. $\forall p\ \exists c\ \forall d\ .\ time_{univ}(p,d) \leq c \cdot time_p(d)$ (program-dependent overhead)
3. $\exists c\ \forall p\ \forall d\ \ .\ time_{univ}(p,d) \leq c \cdot time_p(d)$ (program-independent overhead)

One might expect Overhead 2 in practice, reasoning that if an interpreter *univ* simulates p one step at a time, then $t' \leq f(p) \cdot t$ for some function f. If so, then the interpretation overhead may depend on the program being interpreted, but not on the current input data d.

Unfortunately this is not always so. One counterexample is Turing's original universal machine [30]. Because of the 1-dimensional tape, simulation of the effect of one quintuple in program p may require that the interpreter scans *from* the tape area where p's program code is written, *to* the area where p's currently scanned data square is found, and then *scans back again* to p's program code. Worst-case: $time_{univ}(p,d)$ is larger than p's running time multiplied by the entire size of of its run-time data area.

The same problem appears in most published universal machines, including the TRM model. The problem is the need to "pack" all the simulated program p's data values into one of *univ*'s data values. Applied to TRMs: although *univ* has only a fixed number of registers, there exists no limit to the number of registers that an interpreted program p may have. The root of this problem is that *a limit is inherited from the interpreter*, e.g., the number of registers. Mogensen describes this problem of inherited limits in general terms and with many specific instances in [23].

Is this a problem? Yes (from this paper's viewpoint) since a self-interpreter is needed for most applications of the recursion theorem (beyond self-reproduction and self-recognition).

Smaller overheads have been obtained for some computation models. Interpretation overhead 2 is typical for interpreters with tree-structured data and constant-time pointer access, e.g., interpreters expressed in SCHEME, PROLOG, etc., and the λ-calculus. The partial evaluation literature (overviewed in [15]) contains many such self-interpreters. Mogensen [24] has detailed analyses of the costs of λ-calculus self-interpretation under several execution models.

Overhead 3 is seen in [12] for a very limited language (with one-atom trees and one-variable programs); and for the λ-calculus using some of Mogensen's models and cost measures [24]. The first assumes constant-time pointer access, and the second assumes constant-time variable access (or does not count it). Overhead 3 also holds for the biologically motivated "blob" computation model [10] which has 2-way bonds, no variables, bounded "fan-in" among data values, and a single 2-way activation bond between program and data (which must always be adjacent).

4.4 Futamura projections: partial evaluation can remove interpretation overhead

Partial evaluation concerns *efficient implementation* of the *S-m-n* property. A partial evaluator is simply an *S-1-1* program s^1_1 as in the Rogers axioms. Supposing *univ* is a universal program, the following properties (due to Futamura 1971 [6]) are easy to verify from the definition of *univ* and s^1_1. The first line asserts that a partial evaluator can compile a program *source* into a semantically equivalent program *target*. The second line says that a partial evaluator can generate a compiler; and the third, that a partial evaluator can generate a compiler generator.

Definitions				**Properties**		
1.	$target$	$:=$	$[\![s^1_1]\!](univ, source)$	$\forall d\ .\ [\![target]\!](d)$	$=$	$[\![source]\!](d)$
2.	$compiler$	$:=$	$[\![s^1_1]\!](s^1_1, univ)$	$target$	$=$	$[\![compiler]\!](source)$
3.	$cogen$	$:=$	$[\![s^1_1]\!](s^1_1, s^1_1)$	$compiler$	$=$	$[\![cogen]\!](univ)$

The Futamura projections involve program self-application, but in a way different than that used in the proof of Kleene's theorem, e.g., $[\![s^1_1]\!](s^1_1, univ)$ rather than $[\![s^1_1]\!](q,q)$ or $[\![\hat{q}]\!](\hat{q})$.

The Futamura projections were first fully realised on the computer in Copenhagen in 1985; see [15] for details and references. The expensive self-application in the table for $compiler := [\![s^1_1]\!](s^1_1, univ)$ can be avoided by doing another computation that only needs doing once: $cogen := [\![s^1_1]\!](s^1_1, s^1_1)$. After that, an individual compiler can be generated from any interpreter *int* by the significantly faster run $compiler := [\![cogen]\!](int)$. For details, see [15]. A moral: one deep self-application, to construct $cogen := [\![s^1_1]\!](s^1_1, s^1_1)$, can be used in place of single self-applications $compiler := [\![s^1_1]\!](s^1_1, int)$ that are done repeatedly.

We hope that such analogies will lead to a better complexity-theoretic understanding of Kleene's second recursion theorem.

Complexity issues in partial evaluation are fairly well-understood and partial evaluators well-engineered.

An example “optimality” result from [15], for a simple first-order SCHEME-like language:

Theorem Partial evaluation can remove *all interpretation overhead*, meaning

$$time_{target}(d) \leq time_{source}(d)$$

for all data d and a natural self-interpreter *univ*.

The removal of all interpretation overhead has been achieved in practice as well as in theory.

5 Operational aspects of Kleene’s second recursion theorem

In spite of the theorem’s high impact on theory, it is not easy to reason about its efficiency, e.g., time usage. To being with, there are two distinct efficiency questions with rather different answers:

- The time it takes *to construct* p^* *from* p; and
- *The efficiency of the constructed program* p^*, when run on d.

The constructions used to prove Kleene’s theorem are not complex, and do not require the full power of recursion theory to construct program p^* from p.[11]

5.1 Some operational detail

The Moss approach is similar to Kleene’s, but with different “building blocks,” e.g., no *S-m-n* theorem is used. Based on computer experiments:

1. In practice the Moss approach is somewhat faster and simpler than the Kleene approach, but the transformed program p^* works in essentially the same way for both constructions.
2. How p^* works: for a given p,
   - p^* first computes *a copy of itself, including* p.
   - It then runs p on the copy of p^*, together with the data input d.
3. When run, program p^* has been observed to be *large and slow*. Both constructions generate a *rather expensive set-up phase*, to make the copy of p^*, before ever looking at p’s data input d.
4. The generated program p^* may contain *more than one version of* p *and* s^1_1, in plain and code-generating versions. (This has already been seen in Section 3.3.2.)

5.2 Corner cases

Three “corner cases” (a term due to Polya) that may give some insight into operational behavior:

1. First projection: $[\![p]\!](e,d) = e$. In this case p^* is a self-reproducing program, as in Application 1.
2. Second projection: $[\![p]\!](e,d) = d$. By SRT $[\![p^*]\!](d) = [\![p]\!](p^*,d) = d$, so p^* computes the identity function, but slowly. It first constructs a copy of itself and then runs p, ignoring the copy it made.
3. $p = univ$. By SRT $[\![p^*]\!](d) = [\![p]\!](p^*,d) = [\![univ]\!](p^*,d) = [\![p^*]\!](d)$, which makes no restriction at all on p^*. The resulting program p^* loops infinitely.

[11]The complexity of running p^*, i.e.., of computing $[\![p^*]\!](d)$, can, however be high, depending on program p.

5.3 Can more efficient SRT output programs be obtained?

In the special case that *p does not use its first argument* p^*, as in in corner case 2, one would expect

- $time_p(p^*, d) = f(d)$ for some function f
- $time_{p^*}(d) = c + f(d)$ for some constant c (cf. Section 5.1)

These expectations hold in computer experiments, but the constant c is *very large*.

The unexpected exponential time behavior of the factorial example in application 5 could be circumvented as in Section 3.4 and [9], but at considerable cost: *interpretive execution of all programs*. Can this effect be achieved more economically, e.g., by a stronger s^1_1 algorithm?

5.4 Utility of a more efficient program specialiser.

Kleene's proof is based on the *S-m-n* construction, so would be natural to expect the Kleene SRT construction to benefit from using a state-of-the-art partial evaluator, e.g., as described in [15].

Bonfante (continuing the line of [2]) added to the end of s^1_1 a simple optimiser: a "dead code" detector and eliminator. This was enough to eliminate all the unnecessary computation seen in corner case 2. It is less clear, however, where such optimisations could be put into the Moss construction.

5.5 Relating the `1#` and TINY SRT constructions

5.5.1 Some experiments with `1#`.

The results reported by Moss in [26] led this paper's author to develop a straightforward `1#` implementation in SCHEME, with a step counter to evaluate running times. Some comments:

- `1#` is a register machine model, so program $\texttt{move}_{i,j}$ (used to assign $\texttt{R}_i\texttt{:=}\texttt{R}_j$) takes time $O(|R_j|)$.
- Data structures are the main difference between `1#` and TINY. The linear strings in $\{\texttt{1},\texttt{\#}\}^*$ must be scanned one bit at a time, in contrast to TINY's constant-time pointer operations.
- Observed for the Moss SRT construction:
  - Computing $[\![\texttt{write}]\!](x)$ takes time $O(|x|)$, as does $[\![\texttt{diag}]\!](x)$.
  - For a small p, the set-up phase of p^* (computing $[\![\hat{q}]\!](\hat{q})$) takes between 20,000 and 40,000 steps (depending on implementation choices).
  - A significant factor in computing $[\![\hat{q}]\!](\hat{q})$ was the time $[\![\texttt{write}]\!](x)$ and $[\![\texttt{diag}]\!](x)$ used to read x, and to compute their output values while scanning several versions of program p.

We also implemented the Kleene version of the SRT construction in `1#` (specialising by $\texttt{s}^1_1$ as in [26]). It ran about twice as slowly as the Moss SRT construction. Further, experiments were done with a `1#` variant with constant-time assignments; this is natural for programming languages. The resulting Moss SRT construction ran roughly twice as fast as the original `1#`.

5.5.2 The Moss construction in constant time using TINY.

Every TINY program runs in constant time independent of the size of its input, including the Kleene SRT construction seen in Section 3.3 We will not re-do the complete proof of Section 3.5, but just show that central components of the Moss construction are expressible in TINY.

First, program composition $|$: Let $p = \boxed{\texttt{read x; C}_p\texttt{; write y}}$ and $q = \boxed{\texttt{read y; C}_q\texttt{; write z}}$. Without loss of generality, p,q have disjoint variables, except for `y`. TINY has no need for a time-consuming "append" operation, since program

$$p \mid q = \boxed{\texttt{read x; (C}_p\texttt{ ; C}_q\texttt{); write z}}$$

behaves as required, satisfying $[\![p \mid q]\!](x) = [\![q]\!]([\![p]\!](x)$. Expressed in concrete syntax, the program composer should transform program inputs `((x)` $\underline{\texttt{C}_p}$ `y)` and `((y)` $\underline{\texttt{C}_q}$ `z)` into `((x) (;` $\underline{\texttt{C}_p}$ $\underline{\texttt{C}_q}$`) z)`. This is straightforward to program in TINY.

Next, we need a program such that $[\![[\![\texttt{write}]\!](x)]\!]() = x$ for any $x \in D$. For example, $[\![\texttt{write}]\!](x)$ could yield as output the TINY program $w_x = \boxed{\texttt{read; out := 'x; write out}}$. In concrete syntax:

$$w_x = \texttt{(() (:= out (QUOTE } x\texttt{) out)}$$

A program `write` to generate w_x from x:

write =
```
read x;
out := list('(), (list ':=, 'out, list ('QUOTE, x)), 'out);
write out
```

A program `diag` satisfying $[\![[\![\texttt{diag}]\!](r)]\!]() = [\![r]\!](r)$, for $r \in Pgms$. Goal: $[\![\texttt{diag}]\!](r)$ is a program d_r such that $[\![d_r]\!]() = [\![r]\!](r)$. Let $r = \boxed{\texttt{read x; C}_r\texttt{; write out}}$ have concrete syntax `((x)` $\underline{\texttt{C}_r}$ `out)`. Then d_r could be

$$d_r = \boxed{\texttt{read; x := 'r; C}_r\texttt{; write out}}.$$

or, in concrete syntax:

$$d_r = \texttt{(() (; (:= x (QUOTE r))} \ \underline{\texttt{C}_r}\texttt{) out)}$$

A program `diag` to generate d_r from r:

diag =
```
read r;
inputvar      :=   hd hd r; C := hd tl r; outputvar := hd tl tl r;
initialise    :=   list(':=, inputvar, list('QUOTE, r) );
body          :=   list('; ,initialise, C);
outpgm        :=   list(tl hd pgm, body, outputvar);
write outpgm
```

This `diag` is just TINY specialiser s^1_1, modified to generate code that first copies (a pointer to) static data input `r`, to build `x := 'r`; followed by its body $\texttt{C}_r$. The generated code's net effect is to execute program r on input r.

We omit the similar but tedious details of building a TINY version of program $\hat{q}$. The "append" effect of $\texttt{move}_{i,j}$ is achieved in constant time by TINY's "`;`", as used above for $|$.

6 Related work, future work, and acknowledgements

6.1 Related work

Kleene's second recursion theorem attracted interest since first published in 1938, shortly after Turing's pathbreaking 1936 work that founded computability theory [30]. Kleene's apparently quite theoretical result has shown a staying power in areas far beyond its frequent usage by recursion theorists, as well-documented by Moschovakis in 2010 [25]. One reason for such widespread interest is the way it is proven: in essence Kleene constructed a self-reproducing program. This is particularly surprising since Kleene did this in a quite constructive way in the 1930s, long before the first computer was built.

Computer scientists have for many years repeatedly re-discovered and re-solved the goal of building self-reproducing programs, cf. an elegant example by Thompson [29]. John Case applied this fascinating theorem in both recursive function theory and in computer learning, e.g., see [4]. A group at Nancy led by Jean-Yves Marion has related Kleene's theorem to computer viruses [3], and devised TINY.

Our own early interest in the theory-practice interface led to a 1989 paper [9]. While that solution worked, a drawback was that its usage always involved at least one level of interpretation overhead.

Since then the 2006 work by Lawrence Moss [26] brought Kleene's result strongly to the attention of computer scientists. Further, Oleg Kiselyov [16] has recently worked on the problem from a functional programming viewpoint, a starting point being an unusually efficient self-interpreter written in the λ-calculus using a higher order representation of program syntax. The work in this paper involves much lower-level languages than the ones used by Kiselyov and Mogensen [16, 24].

6.2 Future work

This paper's results concern mostly the time to produce p^* by the Kleene and Moss constructions, but very little has been said about the runtime efficiency of computing $[\![p^*]\!](d)$ for Turing-complete languages (beyond mentioning that this question motivated [9]). Here it becomes interesting, since straightforwardly applying the Kleene or Moss constructions often gives *unnaturally inefficient solutions*, e.g., the nested self-interpretations seen in the recursion example of Application 5.

Following are some questions and goals for future work.

1. A question concerning the Moss 2-stage SRT construction: can an efficient specialiser (e.g., as in [15]) be usefully applied to the result of `diag` from Section 5.5?

2. To what extent can an efficient specialiser be used to produce better fixpoint programs p^*?

3. It is natural to ask whether nested self-interpretation can be avoided without adding an interpretive overhead to every program execution (as seen in [9]).

4. Investigate the changes to the SRT efficiency results if one supposes the implementation language has a self-interpreter *with only additive overhead*, meaning:

$$\forall p \; \exists c \; \forall d \; . \; time_{univ}(p,d) \leq c + time_p(d)$$

   This can be done if (1) programs are data; and (2) one has an instruction with the effect of "goto data". An interpreter *univ* could first compile its input program p to the language in which the interpreter itself is written; and then jump, i.e., transfer control, to this code's first instruction. Such effects can be achieved in the von Neumann computation model (although finite), in Marion's SRM model extending the Moss `1#` language [20], and in a planned modest extension of the Blob model [10].

5. In a Turing-complete language, running times may of course be much larger than the constant-time bounds of TINY. If program p may contain control loops, challenging problems include:
   - how to find bounds on $time_{p^*}(d)$ as built by existing constructions; and
   - how to achieve better running times by new constructions.

These questions could be approached pragmatically, or computation-theoretically.

6. A more general problem: find relations between the self-application from Kleene's SRT, e.g., $[\![s^1_1]\!](q,q)$ or $[\![\hat{q}]\!](\hat{q})$; and the self-application used in the Futamura projections, e.g., $[\![s^1_1]\!](s^1_1, univ)$ or $[\![s^1_1]\!](s^1_1, s^1_1)$.

7. Finally, suppose p is "extensional", as in Rice's Theorem: the output value $[\![p]\!](e,x)$ depends only on the semantics of argument e, so $[\![e]\!] = [\![e']\!]$ implies $\forall x\ [\![p]\!](e,x) = [\![p]\!](e',x)$. Operationally, all that can be done with p's program argument e is to run it (perhaps nested, e.g., $[\![e]\!]([\![e]\!](x))$.

   Somehow this seems close to Kleene's *first recursion theorem*. Can a precise connection be made? A gap to be closed is that the first recursion theorem concerns computable functionals (second-order), rather than first-order functions.

6.3 Acknowledgements

Thanks to NII (Tokyo), IMDEA Software (Madrid), UTS (Sydney), and the COLA project (University of Copenhagen) for good environments to do this work. A 2013 visit to LORIA (Nancy, France) to work with Jean-Yves Marion and Guillaume Bonfante was a great help, both in focusing this paper, and in thinking about future exciting directions, e.g., self-interpretation with only additive overhead. Thanks for comments on form and content to anonymous referees and to Geoff Hamilton, Barry Jay, Torben Mogensen, Lawrence Moss, Jean-Yves Moyen and Jakob Grue Simonsen.

Finally and on a broader plane, I would like *to thank Dave Schmidt* for numerous wide-ranging and deep discussions of programmming languages and their semantics and implementation. These have occurred over many years since our first contact in Kansas in the 1970s and subsequent years in Aarhus and Copenhagen.

References

[1] Manuel Blum (1967): *A Machine-Independent Theory of the Complexity of Recursive Functions*. *Journal of the Association for Computing Machinery (JACM)* 14(2), pp. 322–336. Available at `http://doi.acm.org/10.1145/321386.321395`.

[2] Guillaume Bonfante & Benjamin Greenbaum (2013): *Immune Systems in Computer Virology (working notes)*. Technical Report, Université de Lorraine, Nancy, France.

[3] Guillaume Bonfante, Matthieu Kaczmarek & Jean-Yves Marion (2007): *A Classification of Viruses Through Recursion Theorems*. In: *Computability in Europe*, *Lecture Notes in Computer Science* 4497, Springer, pp. 73–82. Available at `http://dx.doi.org/10.1007/978-3-540-73001-9_8`.

[4] John Case & Samuel E. Moelius (2012): *Program Self-Reference in Constructive Scott Subdomains*. *Theory of Computing Systems* 51(1), pp. 22–49. Available at `http://dx.doi.org/10.1007/s00224-011-9372-1`.

[5] Alonzo Church & J. Barkley Rosser (1936): *Some Properties of Conversion*. *Transactions of the American Mathematical Society* 39, pp. 11–21. Available at `http://dx.doi.org/10.2307/1989762`.

[6] Yoshiko Futamura (1999): *Partial evaluation of computing process – an approach to a compiler-compiler*. *Higher-Order and Symbolic Computation* 12(4), pp. 381–391. Available at `http://dx.doi.org/10.1023/A:1010095604496`.

[7] Robin Gandy (1988): *The Confluence of Ideas in 1936*. In Herken [11], pp. 55–112. Available at `http://dx.doi.org/10.1007/978-3-7091-6597-3_3`.

[8] Roberto Giacobazzi, Neil D. Jones & Isabella Mastroeni (2012): *Obfuscation by partial evaluation of distorted interpreters*. In: *ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012*, pp. 63–72. Available at `http://doi.acm.org/10.1145/2103746.2103761`.

[9] Torben Amtoft Hansen, Thomas Nikolajsen, Jesper Larsson Träff & Neil D. Jones (1989): *Experiments with Implementations of Two Theoretical Constructions*. In: *Logic at Botik*, *LNCS* 363, Springer, pp. 119–133. Available at `http://dx.doi.org/10.1007/3-540-51237-3_11`.

[10] Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen & Søren Bjerregaard Vrist (2011): *Programming in Biomolecular Computation: Programs, Self-Interpretation and Visualisation*. *Scientific Annals of Computer Science* 21(1), pp. 73–106. Available at `http://www.infoiasi.ro/bin/Annals/Article?v=XXI1{&}a=9`.

[11] Rolf Herken, editor (1988): *The Universal Turing Machine. A Half-Century Survey*. Oxford University Press. Available at `http://dx.doi.org/10.1007/978-3-7091-6597-3`.

[12] Neil D. Jones (1993): *Constant time factors do matter*. In: *ACM Symposium on Theory of Computing, STOC 1993*, ACM, pp. 602–611. Available at `http://doi.acm.org/10.1145/167088.167244`.

[13] Neil D. Jones (1997): *Computability and Complexity from a Programming Perspective*, 1 edition. *Foundations of Computing* , MIT Press, Boston, London. Available at `http://dx.doi.org/10.1016/S1571-0661(04)00019-2`.

[14] Neil D. Jones (1999): LOGSPACE *and* PTIME *Characterized by Programming Languages*. *Theoretical Computer Science* 228(1-2), pp. 151–174. Available at `http://dx.doi.org/10.1016/S0304-3975(98)00357-0`.

[15] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

[16] Oleg Kiselyov (2011): *Investigations into Kleene's 2nd recursion theorem*. Technical Report, Fleet Numerical Meteorology and Oceanography Center, `http://okmij.org/ftp/Haskell/Kleene.hs`.

[17] Stephen Cole Kleene (1938): *On Notation for Ordinal Numbers*. *Journal of Symbolic Logic* 3(4), pp. 150–155. Available at `http://dx.doi.org/10.2307/2267778`.

[18] Stephen Cole Kleene (1952): *Introduction to Metamathematics*. Van Nostrand. Available at `http://dx.doi.org/10.1007/978-0-8176-4769-8_11`.

[19] Michael Machtey & Paul Young (1978): *An introduction to the general theory of algorithms*. Theory of computation series, North-Holland, New York.

[20] Jean-Yves Marion (2012): *From Turing machines to computer viruses*. In: *Philosophical transactions of the Royal Society A, Lecture Notes in Computer Science* 370.1971, Royal Society publishing, pp. 3319–3339. Available at `http://dx.doi:10.1098/rsta.2011.0332`.

[21] John McCarthy (1960): *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. *Communications of the Association for Computing Machinery (CACM)* 3(4), pp. 184–195. Available at `http://doi.acm.org/10.1145/367177.367199`.

[22] Marvin Minsky (1967): *Computation: Finite and Infinite Machines*. Prentice-Hall Series in Automatic Computation.

[23] Torben Æ. Mogensen (1996): *Evolution of partial evaluators: removing inherited limits*. In: *Partial Evaluation. Proceedings, LNCS 1110, 303321*, Springer-Verlag, pp. 303–321. Available at `http://dx.doi.org/10.1007/3-540-61580-6_15`.

[24] Torben Æ. Mogensen (2000): *Linear-Time Self-Interpretation of the Pure Lambda Calculus*. *Higher-Order and Symbolic Computation* 13(3), pp. 217–237. Available at `http://dx.doi.org/10.1023/A:1010058213619`.

[25] Yiannis N. Moschovakis (2010): *Kleene's amazing Second Recursion Theorem*. *Bulletin of Symbolic Logic* 16(2), pp. 189–239. Available at `http://dx.doi.org/10.2178/bsl/1286889124`.

[26] Lawrence S. Moss (2006): *Recursion Theorems and Self-Replication Via Text Register Machine Programs*. *Bulletin of the European Association for Theoretical Computer Science* 89, pp. 171–182.

[27] Hartley Rogers (1987): *Theory of recursive functions and effective computability (Reprint from 1967)*. MIT Press. Available at `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2{&}tid=3182`.

[28] John C. Shepherdson & Howard E. Sturgis (1961): *Computability of Recursive Functions*. *Journal of the Association for Computing Machinery (JACM)* 10, pp. 217–255. Available at `http://dx.doi.org/10.1145/321160.321170`.

[29] Ken Thompson (1984): *Reflections on trusting trust*. *Communications of the Association for Computing Machinery (CACM)* 27(8), pp. 761–763. Available at `http://doi.acm.org/10.1145/358198.358210`.

[30] Alan M. Turing (1936-7): *On Computable Numbers with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42(2), pp. 230–265.