Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq

Yannick Forster and Gert Smolka
Saarland University

Saarland University, Saarbrücken, Germany
{forster,smolka}@ps.uni-saarland.de

Abstract. We formalise a weak call-by-value λ-calculus we call L in the constructive type theory of Coq and study it as a minimal functional programming language and as a model of computation. We show key results including (1) semantic properties of procedures are undecidable, (2) the class of total procedures is not recognisable, (3) a class is decidable if it is recognisable, corecognisable, and logically decidable, and (4) a class is recognisable if and only if it is enumerable. Most of the results require a step-indexed self-interpreter. All results are verified formally and constructively, which is the challenge of the project. The verification techniques we use for procedures will apply to call-by-value functional programming languages formalised in Coq in general.

1 Introduction

We study a minimal functional programming language L realising a subsystem of the λ-calculus [3] known as weak call-by-value λ-calculus [8]. As in most programming languages, β-reduction in weak call-by-value λ-calculus is only applicable if the redex is not below an abstraction and if the argument is an abstraction. Our goal is to formally and constructively prove the basic results from computability theory [9,11] for L. The project involves the formal verification of self-interpreters and other procedures computing with encodings of procedures. The verification techniques we use will apply to call-by-value functional programming languages formalised in Coq in general. We base our work on the constructive type theory of Coq [15] and provide a development verifying all results.

The results from computability theory we prove for L include (1) semantic properties of procedures are undecidable (Rice's theorem), (2) the class of total procedures is not recognisable, (3) a class is decidable if it is recognisable, corecognisable, and logically decidable (Post's theorem), and (4) a class is recognisable if and only if it is enumerable.

We prove that procedural decidability in L implies functional decidability in Coq. The converse direction cannot be shown in Coq since Coq is consistent with the assumption that every class is functionally decidable and procedurally undecidable classes always exist. The same will be true for any Turing-complete model of computation formalised in Coq.

The result that procedural decidability implies functional decidability seems contradictory at first since procedures come with unguarded recursion while functions are confined to guarded recursion. The apparent paradox disappears once one realises that procedural decidability means that termination of a decision procedure can be shown in Coq's constructive type theory.

Comparing L with the full λ-calculus, we find that L is more realistic as a programming language and simpler when it comes to semantics and program verification. The restrictions L imposes on β-reduction eliminate the need for capture-free substitution and provide for a uniform confluence property [13,8] ensuring that all evaluating reduction sequences of a term have the same length. Uniform confluence simplifies the construction and verification of a self-interpreter by eliminating the need for a reduction strategy like leftmost-outermost. Moreover, uniform confluence for L is easier to prove than confluence for the full λ-calculus.

While L simplifies the full λ-calculus, it inherits powerful techniques developed for the λ-calculus: Procedural recursion can be expressed with self-application, inductive data types can be expressed with Scott encodings [12,10], and program verification can be based on one-step reduction, the accompanying equivalence, and the connecting Church-Rosser property.

One place where the commitment to a constructive theory prominently shows is Post's theorem. The classical formulation of Post's theorem states that a class is decidable if it is recognisable and corecognisable. The classical formulation of Post's theorem is equivalent to Markov's principle and does not hold in a purely constructive setting [7]. We show Post's theorem with the extra assumption that the class is logically decidable. The extra assumption is needed so that we can prove termination of the procedure deciding the class. We refer to the classical formulation of Post's theorem for L as Markov's principle for L and establish two complementary characterisations.

Related Work. There is not much work on computability theory in constructive type theory. We are aware of Asperti and Ricciotti [1,2] who formalise Turing machines in Matita including a verified universal machine and a verified reduction of multi-tape machines to single-tape machines. They do not consider decidable and recognisable classes. Ciaffaglione [6] formalises Turing machines coinductively in Coq and shows the agreement between a big-step and a small-step semantics.

Bauer [4] develops a constructive and anti-classical computability theory abstracting away from concrete models of computation.

There is substantial work on computability theory in classical higher-order logic. Norrish [14] presents a formal development of computability theory in HOL4 where he considers full λ-calculus and partial recursive functions and proves their computational equivalence. Norrish studies decidable and recognisable classes, verifies self-interpreters, and proves basic results including the theorems of Rice and Post.

There are substantial differences between our work and Norrish [14] apart from the fact that Norrish works in a classical setting. Following Barendregt [3], Norrish works with full λ-calculus and Gödel-Church encodings. We work with L

and Scott encodings instead. Church encodings are not possible using weak β-reduction. We remark that Scott encodings are simpler than Gödel-Church encodings (since they don't involve recursion). Norrish proves Rice's theorem for partial recursive functions while we prove the theorem directly for procedures in L.

Xu, Zhang, and Urban [16] formalise Turing machines, abacus machines, and partial recursive functions in Isabelle (classical higher-order logic) and show their computational equivalence following Boolos et al. [5]. They prove the existence of a universal function. They do not consider the theorems of Rice and Post.

Dal Lago and Martini [8] consider a weak call-by-value λ-calculus and show that Turing machines and procedures in the calculus can simulate each other with polynomial-time overhead, thus providing evidence that a weak call-by-value λ-calculus may serve as a reasonable complexity model. Their λ-calculus is different from ours in that it employs full substitution and β-reduction is possible if the argument is a variable. Like us, they use Scott encodings of data types. Their work is not formalised.

Main Contributions. Our work is the first formal study of weak call-by-value λ-calculus covering both language semantics and program verification. We are also first in proving results from computability theory for a programming language in constructive type theory.

The development of this paper is carried out in constructive type theory and outlines a machine-checked Coq development. The Coq development is surprisingly compact and consists of less than 2000 lines of code. The theorems in the pdf of the paper are hyperlinked with their formalisations in the Coq development, which can be found at `http://www.ps.uni-saarland.de/extras/L-computability`.

2 Specification

We start by specifying essential properties of the functional language L we will work with and by describing main results from computability theory we will prove for L.

We assume a discrete type of *terms* and a class of terms called *procedures*. We will use the letters s, t, u, v, w for terms and the letters p, q for classes of terms.

We assume a functional relation $s \rhd t$ on terms called *evaluation*. We say that a term s *evaluates* and write $\mathcal{E}s$ if there is a term t such that $s \rhd t$.

We assume a function st from terms to terms called *application*.

We assume two procedures T and F such that $\mathrm{T} \neq \mathrm{F}$ and $\mathrm{T}st \rhd s$ and $\mathrm{F}st \rhd t$ for all procedures s, t. As usual, we omit parentheses in nested applications; for instance, $\mathrm{T}st$ stands for $(\mathrm{T}s)t$.

We assume an injective function $\overline{s}$ from terms to procedures called *term encoding*. The purpose of the encoding function is to encode a term into a procedure providing the term as data to other procedures. This is a subtle point that will

become clear later. For now it suffices to know that $\bar{s}$ is a function from terms to procedures.

We now define *decidable*, *recognisable*, and *corecognisable* classes of terms:

- A procedure u *decides* a class p if $\forall s.\ ps \land u\bar{s} \rhd \mathrm{T} \lor \neg ps \land u\bar{s} \rhd \mathrm{F}$.
- A procedure u *recognises* a class p if $\forall s.\ ps \leftrightarrow \mathcal{E}(u\bar{s})$.
- A procedure u *corecognises* a class p if $\forall s.\ \neg ps \leftrightarrow \mathcal{E}(u\bar{s})$.

Our assumptions suffice to establish the existence of undecidable and unrecognisable classes.

Fact 1. *Let u decide p. Then $ps \leftrightarrow u\bar{s} \rhd \mathrm{T}$ and $\neg ps \leftrightarrow u\bar{s} \rhd \mathrm{F}$.*

Fact 2. *$\lambda s.\neg(s\bar{s} \rhd \mathrm{T})$ is not decidable, and $\lambda s.\neg\mathcal{E}(s\bar{s})$ is not recognisable.*

Proof. Suppose u decides $\lambda s.\neg(s\bar{s} \rhd \mathrm{T})$. Then $u\bar{s} \rhd \mathrm{T} \leftrightarrow \neg(s\bar{s} \rhd \mathrm{T})$ for all s. The equivalence is contradictory for $s := u$. The proof for the unrecognisable class is similar. □

We need different notions of decidability in this paper. We call a class p

- *logically decidable* if there is a proof of $\forall s.\ ps \lor \neg ps$.
- *functionally decidable* if there is a function f such that $\forall s.\ ps \leftrightarrow fs = \mathsf{true}$.
- *procedurally decidable* if there is a procedure deciding p.

If we say decidable without further qualification, we always mean procedurally decidable. Note that functionally decidable classes are logically decidable.

We define two semantic properties of terms. A term s is *total* if the application $s\bar{t}$ evaluates for every term t. *Semantic equivalence* of terms is defined as $s \approx t := \forall uv.\ s\bar{u} \rhd v \leftrightarrow t\bar{u} \rhd v$. Note that if $s \approx t$, then s is total iff t is total.

We can now specify major results we will prove in this paper.

- *Rice's theorem.* Every nontrivial class of procedures that doesn't distinguish between semantically equivalent procedures is undecidable.
- *Modesty.* Procedurally decidable classes are functionally decidable.
- *Totality.* The class of total procedures is unrecognisable.
- *Post's Theorem.* A class is decidable if it is recognisable, corecognisable, and logically decidable.

We will also consider enumerable classes and show that they agree with recognisable classes. All results but Rice's theorem require a step-indexed interpreter or step-indexed self-interpreter.

Note the distinction between functions and procedures. While functions are entities of the typed specification language (i.e., Coq's type theory), procedures are entities of the untyped programming language L formalised in the specification language by means of a deep embedding. As we will see, L comes with unbounded recursion and thus admits nonterminating procedures. In contrast, Coq's type theory is designed such that functions always terminate.

3 Definition of L

We will work with the terms of the λ-calculus. We restrict β-reduction such that β-redexes can only be reduced if (1) they are not within an abstraction and (2) their argument term is an abstraction. With this restriction the terms $\lambda x.(\lambda y.y)(\lambda y.y)$ and $(\lambda x.x)x$ are irreducible. We speak of *weak call-by-value β-reduction* and write $s \succ t$ if t can be obtained from s with a single weak call-by-value β-reduction step. We will define the evaluation relation such that $s \rhd t$ holds iff $s \succ^* t$ and t is an abstraction. Procedures will be defined as closed abstractions.

Since we want formal proofs we are forced to formally define the concrete weak call-by-value λ-calculus L we are working with. In fact, there are some design choices. We will work with de Bruijn terms and capturing substitution, two design decisions providing for a straightforward formal development.

We start the formal definition of L with an inductive type of *terms*:

$$s, t \ ::= \ n \mid st \mid \lambda s \qquad (n : \mathbf{N})$$

We fix some terms for further use:

$\mathrm{I} = \lambda x.x$	$\mathrm{T} = \lambda xy.x$	$\mathrm{F} = \lambda xy.y$	$\omega = \lambda x.xx$	$\mathrm{D} = \lambda x.\omega\omega$
$:= \lambda 0$	$:= \lambda(\lambda 1)$	$:= \lambda(\lambda 0)$	$:= \lambda(00)$	$:= \lambda(\omega\omega)$

For readability, we will usually write concrete terms with named abstractions, as shown above. The Coq development provides a function translating named abstraction to the implementation using de Bruijn indices. Note that D is reducible in the full λ-calculus but will not be reducible in L.

We define a *substitution function* s_u^k that replaces every free occurrence of a variable k in a term s with a term u. The definition is by recursion on s:

$$\begin{aligned} n_u^k &= \mathsf{if}\ n{=}k\ \mathsf{then}\ u\ \mathsf{else}\ n \\ (st)_u^k &= (s_u^k)(t_u^k) \\ (\lambda s)_u^k &= \lambda(s_u^{Sk}) \end{aligned}$$

A substitution s_u^k may capture free variables in u. Capturing will not affect our development since it doesn't affect confluence and our results mostly concern closed terms.

We now give a formal definition of closed terms. Closed terms are important for our development since procedures are defined as closed abstractions and substitutions do not affect closed terms. Moreover, we need a decider for the class of closed terms.

We define a recursive boolean function *bound* $k\, s$ satisfying the equations

$$\begin{aligned} \mathit{bound}\ k\ n &= \mathsf{if}\ n{<}k\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false} \\ \mathit{bound}\ k\ (st) &= \mathsf{if}\ \mathit{bound}\, k\, s\ \mathsf{then}\ \mathit{bound}\, k\, t\ \mathsf{else}\ \mathsf{false} \\ \mathit{bound}\ k\ (\lambda s) &= \mathit{bound}\ (Sk)\ s \end{aligned}$$

Speaking informally, *bound* $k\,s$ tests whether every free variable in s is smaller than k. We say that s is *bound* by n if *bound* $n\,s = \mathsf{true}$. We now define *closed* terms as terms bound by 0, and *procedures* as closed abstractions. Note that the terms I, T, F, ω, and D are all procedures. The following fact will be used tacitly in many proofs.

Fact 3. *If s is bound by n and $k \geq n$, then $s^k_u = s$. Moreover, $s^k_u = s$ for closed s.*

We define *evaluation* $s \rhd t$ as an inductive predicate:

$$\frac{}{\lambda s \rhd \lambda s} \qquad \frac{s \rhd \lambda u \qquad t \rhd v \qquad u^0_v \rhd w}{st \rhd w}$$

Recall that we write $\mathcal{E}s$ and say that s *evaluates* if $s \rhd t$ for some term t.

Fact 4. *1. If $s \rhd t$, then t is an abstraction.*
2. If $s \rhd t$ and s is closed, then t is closed.
3. If st evaluates, then both s and t evaluate.
4. Fst evaluates if and only if both s and t evaluate.
5. $\omega\omega$ does not evaluate.
6. Ds does not evaluate.

4 Uniformly Confluent Reduction Semantics

To provide for the verification of procedures in L, we complement the big-step semantics obtained with the evaluation predicate with a uniformly confluent reduction semantics.

We define one-step *reduction* $s \succ t$ as an inductive predicate:

$$\frac{}{(\lambda s)(\lambda t) \succ s^0_{\lambda t}} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

We also define two reduction relations $s \succ^* t$ and $s \succ^n t$ as inductive predicates:

$$\frac{}{s \succ^* s} \qquad \frac{s \succ u \qquad u \succ^* t}{s \succ^* t} \qquad \frac{}{s \succ^0 s} \qquad \frac{s \succ u \qquad u \succ^n t}{s \succ^{Sn} t}$$

Fact 5. *1. $s \succ^* t$ is transitive.*
2. If $s \succ^* s'$ and $t \succ^* t'$, then $st \succ^* s't'$.
3. $s \succ^* t$ iff $s \succ^n t$ for some n.
4. If $s \succ^m s'$ and $s' \succ^n t$, then $s \succ^{m+n} t$.
5. If $s \rhd t$, then $s \succ^* t$ and t is an abstraction.
6. If $s \succ s'$ and $s' \rhd t$, then $s \rhd t$.
7. If $s \succ^* s'$ and $s' \rhd t$, then $s \rhd t$.

8. *If $s \succ^* t$ and t is an abstraction, then $s \rhd t$.*

With the reduction semantics we can specify procedural recursion, which is essential for our goals.

Fact 6. (Recursion Operator) *There is a function ρ from terms to terms such that (1) ρs is a procedure if s is closed and (2) $(\rho u)v \succ^3 u(\rho u)v$ for all procedures u and v.*

Proof. $\rho s := \lambda x.CCsx$ with $C := \lambda xy.y(\lambda z.xxyz)$ does the job. □

We call the function ρ *recursion operator* since it provides for recursive programming in L using well-known techniques from functional programming.

The weak call-by-value λ-calculus in general and L in particular enjoy a strong confluence property [13,8] we call uniform confluence.

Fact 7. (Uniform Confluence)

1. *If $s \succ t_1$ and $s \succ t_2$, then either $t_1 = t_2$ or $t_1 \succ u$ and $t_2 \succ u$ for some u.*
2. *If $s \succ^m t_1$ and $s \succ^n t_2$, then there exist numbers $k \le n$ and $l \le m$ and a term u such that $t_1 \succ^k u$ and $t_2 \succ^l u$ and $m + k = n + l$.*

Corollary 8. *$s \succ t$ is confluent.*

We define $s \rhd^n t := s \succ^n t \wedge \textit{abstraction } t$ and $s \succ^+ t := \exists s'.\, s \succ s' \wedge s' \succ^* t$.

Corollary 9. (Unique Step Index) *If $s \rhd^m t$ and $s \rhd^n t$, then $m = n$.*

Corollary 10. (Triangle) *If $s \rhd^n t$ and $s \succ^+ s'$, then $s' \rhd^k t$ for some $k < n$.*

We define *reduction equivalence* $s \equiv t$ as the equivalence closure of reduction:

$$\frac{s \succ t}{s \equiv t} \qquad \frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \qquad t \equiv u}{s \equiv u}$$

Reduction equivalence enjoys the usual Church-Rosser properties and will play a major role in the verification of procedures.

Fact 11. (Church-Rosser Properties)

1. *If $s \succ^* t$, then $s \equiv t$.*
2. *If $s \equiv t$, then $s \succ^* u$ and $t \succ^* u$ for some term u.*
3. *If $s \equiv s'$ and $t \equiv t'$, then $st \equiv s't'$.*
4. *$s \equiv t \leftrightarrow s \succ^* t$ if t is a variable or an abstraction.*
5. *$s \rhd t$ iff $s \equiv t$ and t is an abstraction.*
6. *If $s \equiv t$, then $s \rhd u$ iff $t \rhd u$.*

Proof. Claim 1 follows by induction on $s \succ^* t$. Claim 2 follows by induction on $s \equiv t$ and Corollary 8. Claim 3 follows with Claim 2, Fact 5 (2), and Claim 1. The remaining claims follow with Claim 1 and Claim 2. □

Because L employs call-by-value reduction, a conditional $\mathsf{if}\ u\ \mathsf{then}\ s\ \mathsf{else}\ t$ needs to be expressed as $u(\lambda s)(\lambda t)\mathrm{I}$ in general. We have $\mathrm{T}(\lambda s)(\lambda t)\mathrm{I} \succ^* s$ and $\mathrm{F}(\lambda s)(\lambda t)\mathrm{I} \succ^* t$.

5 Scott Encoding of Numbers

Seen as a programming language, L is a language where all values are procedures. We now show how procedures can encode data using a scheme known as Scott encoding [12,10]. We start with numbers, whose Scott encoding looks as follows:

$$\widehat{0} \;:=\; \lambda ab.a \qquad\qquad \widehat{Sn} \;:=\; \lambda ab.b\,\widehat{n}$$

Note that $\widehat{n}$ is an injective function from numbers to procedures. We have the equivalences

$$\widehat{0}\,s\,t \equiv s \qquad\qquad \widehat{Sn}\,s\,t \equiv t\,\widehat{n}$$

for all evaluable closed terms s, t and all numbers n. The equivalences tell us that the procedure $\widehat{n}$ can be used as a match construct for the encoded number n.

We define a procedure $\text{Succ} := \lambda xab.bx$ such that $\text{Succ}\,\widehat{n} \equiv \widehat{Sn}$. Note that the procedures $\widehat{0}$ and Succ act as the constructors of the Scott encoding of numbers.

Programming with Scott encodings is convenient in that we can follow familiar patterns from functional programming. We demonstrate the case with a *functional specification*

$$\forall mn.\;\; \text{Add}\;\widehat{m}\;\widehat{n} \;\equiv\; \widehat{m+n}$$

of a procedure Add for addition. We say that we are looking for a procedure Add *realising* the addition function $m + n$. A well-known *recursive specification* for the addition function consists of the quantified equations $0 + n = n$ and $Sm + n = S(m + n)$. This gives us a recursive specification for the procedure Add (quantification of m and n is omitted):

$$\text{Add}\;\widehat{0}\;\widehat{n} \equiv \widehat{n} \qquad\qquad \text{Add}\;\widehat{Sm}\;\widehat{n} \equiv \text{Succ}\,(\text{Add}\;\widehat{m}\;\widehat{n})$$

With induction on m one can now show that a procedure Add satisfies the functional specification if it satisfies the recursive specification. The recursive specification of Add suggest a recursive definition of Add using L's recursion operator ρ:

$$\text{Add} := \rho(\lambda xyz.yz(\lambda y_0.\text{Succ}(xy_0z)))$$

Using the equivalences for the recursion operator ρ and those for the procedures $\widehat{0}$, $\widehat{Sn}$, and Succ, one easily verifies that Add satisfies the equivalences of the recursive specification. Hence Add satisfies the functional specification we started with.

The functional specification of Add has the virtue that properties of Add like commutativity (i.e., $\text{Add}\;\widehat{m}\;\widehat{n} \equiv \text{Add}\;\widehat{n}\;\widehat{m}$) follow from properties of the addition function $m + n$.

The method we have seen makes it straightforward to obtain a procedure realising a function given a recursive specification of the function. Once we have

Scott encodings for terms and a few other inductive data types, the vast majority of procedures needed for our development can be derived routinely from their functional specifications. We are working on tactics that, given a recursive function, automatically derive a realising procedure and the corresponding correctness lemma.

6 Scott Encoding of Terms

We define the term encoding function $\overline{s}$ specified in Section 2 as follows:

$$\overline{n} := \lambda abc.\, a\, \widehat{n} \qquad \overline{st} := \lambda abc.\, b\, \overline{s}\, \overline{t} \qquad \overline{\lambda s} := \lambda abc.\, c\, \overline{s}$$

This definition agrees with the Scott encoding of the inductive data type for terms. We define the constructors for variables, applications, and abstractions such that they satisfy the equivalences

$$\mathrm{V}\, \widehat{n} \equiv \overline{n} \qquad \mathrm{A}\, \overline{s}\, \overline{t} \equiv \overline{st} \qquad \mathrm{L}\, \overline{s} \equiv \overline{\lambda s}$$

for all numbers n and all terms s and t.

We will define two procedures N and Q satisfying the equivalences

$$\mathrm{N}\, \widehat{n} \equiv \overline{\widehat{n}} \qquad \mathrm{Q}\, \overline{s} \equiv \overline{\overline{s}}$$

for all numbers n and all terms s. The procedure Q is known as *quote* and will be used in the proof of Rice's theorem. The procedure N is an auxiliary procedure needed for the definition of Q. We define the procedures N and Q with the recursion operator realising the following recursive specifications:

$$\begin{aligned} \mathrm{N}\, \widehat{0} &\equiv \overline{\widehat{0}} & \mathrm{Q}\, \overline{n} &\equiv \mathrm{L}(\mathrm{L}(\mathrm{L}(\mathrm{A}\, \overline{2}\, (\mathrm{N}\, \widehat{n})))) \\ \mathrm{N}\, \widehat{Sn} &\equiv \mathrm{L}(\mathrm{L}(\mathrm{A}\, \overline{0}\, (\mathrm{N}\, \widehat{n}))) & \mathrm{Q}\, \overline{st} &\equiv \mathrm{L}(\mathrm{L}(\mathrm{L}(\mathrm{A}(\mathrm{A}\, \overline{1}\, (\mathrm{Q}\, \overline{s}))(\mathrm{Q}\, \overline{t})))) \\ & & \mathrm{Q}\, \overline{\lambda s} &\equiv \mathrm{L}(\mathrm{L}(\mathrm{L}(\mathrm{A}\, \overline{0}\, (\mathrm{Q}\, \overline{s})))) \end{aligned}$$

Given the definitions of procedures N and Q, one first verifies that they satisfy the equivalences of the recursive specifications. Then one shows by induction on numbers and terms that N and Q satisfy the functional specifications we started with. We summarise the results obtained so far.

Fact 12. *There are procedures* V, A, L, *and* Q *such that* $\mathrm{V}\, \widehat{n} \equiv \overline{n}$, $\mathrm{A}\, \overline{s}\, \overline{t} \equiv \overline{st}$, $\mathrm{L}\, \overline{s} \equiv \overline{\lambda s}$, *and* $\mathrm{Q}\, \overline{s} \equiv \overline{\overline{s}}$.

7 Decidable and Recognisable Classes

Now that we have established the term encoding function, we can start proving properties of decidable and recognisable classes. Recall the definitions from Section 2. We will prove the following facts: decidable classes are recognisable; the family of decidable classes is closed under intersection, union, and complement; and the family of recognisable classes is closed under intersection. We establish these facts constructively with translation functions.

Fact 13. *Let u decide p and v decide q. Then:*

1. $\lambda x.ux\,\mathrm{I}\,\mathrm{D}\,\mathrm{I}$ *recognises* p.
2. $\lambda x.ux(vx)\mathrm{F}$ *decides* $\lambda s.ps \wedge qs$.
3. $\lambda x.ux\,\mathrm{T}(vx)$ *decides* $\lambda s.ps \vee qs$.
4. $\lambda x.ux\,\mathrm{F}\,\mathrm{T}$ *decides* $\lambda s.\neg ps$.

Fact 14. $\lambda x.\mathrm{F}(ux)(vx)$ *recognises* $\lambda s.ps \wedge qs$ *if* u *recognise* p *and* v *recognise* q.

We now prove Scott's theorem for L following Barendregt's proof [3] of Scott's theorem for the full λ-calculus. Scott's theorem is useful for proving undecidability of classes that do not distinguish between reduction equivalent closed terms.

Fact 15. *Let s be closed. Then there exists a closed term t such that $t \equiv s\overline{t}$.*

Proof. $t := C\overline{C}$ with $C := \lambda x.s(\mathrm{A}x(\mathrm{Q}x))$ does the job. □

Theorem 16. (Scott)
Every class p satisfying the following conditions is undecidable.

1. *There are closed terms s_1 and s_2 such that ps_1 and $\neg ps_2$.*
2. *If s and t are closed terms such that $s \equiv t$ and ps, then pt.*

Proof. Let p be a class as required and u be a decider for p. Let s_1 and s_2 be closed terms such that ps_1 and $\neg ps_2$. We define $v := \lambda x.ux(\lambda s_2)(\lambda s_1)\,\mathrm{I}$. Fact 15 gives us a closed term t such that $t \equiv v\overline{t} \equiv u\overline{t}(\lambda s_2)(\lambda s_1)\mathrm{I}$. Since u is a decider for p, we have two cases: (1) If $u\overline{t} \equiv \mathrm{T}$ and pt, then $t \equiv s_2$ contradicting $\neg ps_2$; (2) If $u\overline{t} \equiv \mathrm{F}$ and $\neg pt$, then $t \equiv s_1$ contradicting ps_1. □

Corollary 17. *The class of evaluating terms is undecidable.*

Corollary 18. *For every closed term t the class $\lambda s.s \equiv t$ is undecidable.*

8 Reduction Lemma and Rice's Theorem

The reduction lemma formalises a basic result of computability theory and will be used in our proofs of Rice's theorem and the totality theorem. Speaking informally, the reduction lemma says that a class is unrecognisable if it can represent the class $\lambda s.$ *closed* $s \wedge \neg\mathcal{E}(s\overline{s})$ via a procedurally realisable function.

Fact 19. *The class $\lambda s.$ closed $s \wedge \neg\mathcal{E}(s\overline{s})$ is not recognisable.*

Proof. Suppose u is a recogniser for the class. Then $\mathcal{E}(u\overline{u}) \leftrightarrow$ *closed* $u \wedge \neg\mathcal{E}(u\overline{u})$, which is contradictory. □

Fact 20. *There is a decider for the class of closed terms.*

Proof. The decider can be obtained with a procedure realising the boolean function $\mathit{bound}\, k\, s$ defined in Section 3. For this we need a procedure realising a boolean test $m < n$. The construction and verification of both procedures is routine using the techniques from Section 5. □

Lemma 21. (Reduction) *A class p is unrecognisable if there exists a function f such that:*

1. *$p(fs) \leftrightarrow \neg\mathcal{E}(s\overline{s})$ for every closed terms s.*
2. *There is a procedure v such that $v\overline{s} \equiv \overline{fs}$ for all s.*

Proof. Let f be a function satisfying (1) and (2) for a procedure v. Suppose u recognises p. Let C be a recogniser for the class of closed terms (available by Fact 20). We define the procedure

$$w \;:=\; \lambda x.\mathrm{F}(Cx)(u(vx))$$

We have $w\overline{s} \equiv \mathrm{F}(C\overline{s})(u(\overline{fs}))$. Thus $\mathcal{E}(w\overline{s}) \leftrightarrow \mathit{closed}\ s \wedge \mathcal{E}(u(\overline{fs}))$. Since u is a recogniser for p, we have $\mathcal{E}(u(\overline{fs})) \leftrightarrow p(fs)$ for all s. Since $p(fs) \leftrightarrow \neg\mathcal{E}(s\overline{s})$ for closed s by assumption, we have $\mathit{closed}\ s \wedge \mathcal{E}(u(\overline{fs})) \leftrightarrow \mathit{closed}\ s \wedge \neg\mathcal{E}(s\overline{s})$ for all s. Thus w is recogniser for the unrecognisable class of Fact 19. Contradiction. □

We now come to Rice's theorem. Using the reduction lemma, we will first prove a lemma that is stronger than Rice's theorem in that it establishes unrecognisability rather than undecidability. We did not find this lemma in the literature, but for ease of language we will refer to it as Rice's lemma.

Recall the definition of *semantic equivalence*

$$s \approx t \;:=\; \forall uv.\ s\overline{u} \rhd v \leftrightarrow t\overline{u} \rhd v$$

from Section 2. We have $s \equiv t \to s \approx t$ using Fact 11. We say that a class p is *semantic for procedures* if the implication $s \approx t \to ps \to pt$ holds for all procedures s and t.

Lemma 22. (Rice) *Let p be a class that is semantic for procedures such that D is in p and some procedure N is not in p. Then p is unrecognisable.*

Proof. By the reduction lemma. We define fs as a procedure such that for closed s we have $fs \approx D$ if $\neg\mathcal{E}(s\overline{s})$ and $fs \approx N$ if $\mathcal{E}(s\overline{s})$. Here are the definitions of f and the realising procedure v:

$$f := \lambda s.\lambda y.\mathrm{F}(s\overline{s})Ny$$
$$v := \lambda x.\mathrm{L}(\mathrm{A}(\mathrm{A}(\mathrm{A}\,\overline{\mathrm{F}}(\mathrm{A}x(\mathrm{Q}x)))\overline{N})\overline{0})$$

Verifying the proof obligations of the reduction lemma is straightforward. □

Corollary 23. *1. The class of non-total terms is unrecognisable.*

2. *The class of non-total closed terms is unrecognisable.*
3. *The class of non-total procedures is unrecognisable.*

Theorem 24. (Rice) *Every nontrivial class of procedures that is semantic for procedures is undecidable.*

Proof. Let p be a nontrivial class that is semantic for procedures. Suppose p is decidable. We proceed by case analysis for pD.

Let pD. Then p is unrecognisable by Rice's Lemma, contradicting the assumption that p is decidable.

Let $\neg pD$. We observe that $\lambda s.\neg ps$ is semantic for procedures and contains D. Thus $\lambda s.\neg ps$ is unrecognisable by Rice's Lemma, contradicting the assumption that p is decidable. □

Corollary 25. *The class of total procedures is undecidable.*

Rice's theorem looks similar to Scott's theorem but neither can be obtained from the other. Recall that procedures are reduction equivalent only if they are identical.

The key idea in the proof of Rice's lemma is the construction of the procedure v that constructs a procedure that has the right properties. In textbooks this intriguing piece of meta-programming is usually carried out in English using Turing machines in place of procedures. We doubt that there is a satisfying formal proof of Rice's lemma using Turing machines.

9 Step-Indexed Interpreter and Modesty

We will now prove that procedural decidability implies functional decidability. The proof employs a step-indexed interpretation function for the evaluation relation $s \rhd t$. The interpretation function will also serve as the basis for a step-indexed self-interpreter for L, which is needed for the remaining results of this paper.

We use $\mathbf{T}$ to denote the type of terms, $\mathbf{T}_\emptyset$ to denote the option type for $\mathbf{T}$, and $\lfloor s \rfloor$ and $\emptyset$ to denote the values of $\mathbf{T}_\emptyset$. We define a function $\mathit{eval} : \mathbf{N} \to \mathbf{T} \to \mathbf{T}_\emptyset$ satisfying the following recursive specification.

$$\begin{aligned}
\mathit{eval}\ n\ k &= \emptyset \\
\mathit{eval}\ n\ (\lambda s) &= \lfloor \lambda s \rfloor \\
\mathit{eval}\ 0\ (st) &= \emptyset \\
\mathit{eval}\ (Sn)\ (st) &= \mathsf{match}\ \mathit{eval}\ n\ s,\ \mathit{eval}\ n\ t\ \mathsf{with} \\
&\quad \mid\ \lfloor \lambda s \rfloor,\ \lfloor t \rfloor\ \Rightarrow\ \mathit{eval}\ n\ s^0_t \\
&\quad \mid\ _\ _\ \Rightarrow\ \emptyset
\end{aligned}$$

Fact 26. *1. If $\mathit{eval}\ n\ s = \lfloor t \rfloor$, then $\mathit{eval}\ (Sn)\ s = \lfloor t \rfloor$.*

2. If $s \succ s'$ and $\mathit{eval}\ n\ s' = \lfloor t \rfloor$, then $\mathit{eval}\ (Sn)\ s = \lfloor t \rfloor$.

3. $s \rhd t$ if and only if $\mathit{eval}\ n\ s = \lfloor t \rfloor$ for some n.

Proof. Claim 1 follows by induction on n. Claim 2 follows by induction on n using Claim 1. Claim 3, direction $\to$, follows by induction on $s \succ^* t$ and Claim 2. Claim 3, direction $\leftarrow$, follows by induction on n. □

Lemma 27. *There is a function of type* $\forall s.\ \mathcal{E}\,s \to \Sigma t.\ s \rhd t$.

Proof. Let s be a term such that $\mathcal{E}s$. Then we have $\exists nt.\ \mathit{eval}\ n\ s = \lfloor t \rfloor$ by Fact 26. Since the predicate $\lambda n.\ \exists t.\ \mathit{eval}\ n\ s = \lfloor t \rfloor$ is functionally decidable, constructive choice for $\mathbf{N}$ gives us an n such that $\exists t.\ \mathit{eval}\ n\ s = \lfloor t \rfloor$. Hence we have t such that $\mathit{eval}\ n\ s = \lfloor t \rfloor$. Thus $s \rhd t$ with Fact 26. □

Theorem 28. (Modesty) *Procedurally decidable classes are functionally decidable.*

Proof. Let u be a decider for p. Let s be a term. Lemma 27 gives us a term v such that $u\overline{s} \rhd v$. Now we return true if $v = \mathrm{T}$ and false otherwise. □

We can also show modesty results for procedures other than deciders. For this we need a decoding for the Scott encoding of terms.

Fact 29. (Decoding) *There is a function* $\delta : \mathbf{T} \to \mathbf{T}_\emptyset$ *such that (1)* $\delta\,\overline{s} = \lfloor s \rfloor$ *and (2)* $\delta\,s = \lfloor t \rfloor \to \overline{t} = s$ *for all terms* s *and* t.

Fact 30. (Modesty) *Let* u *be a procedure such that* $\forall s \exists t.\ u\,\overline{s} \rhd \overline{t}$. *Then there is a function* $f : \mathbf{T} \to \mathbf{T}$ *such that* $\forall s.\ u\,\overline{s} \rhd \overline{fs}$.

Proof. Follows with Lemma 27 and Fact 29. □

10 Choose

Choose is a procedure that given a decidable test searches for a number satisfying the test. Choose is reminiscent of minimisation for recursive functions [5]. Choose will be the only procedure in our development using truly unguarded recursion. We will use choose to obtain unbounded self-interpreters and to obtain recognisers from enumerators.

A *test* is a procedure u such that for every number n either $u\widehat{n} \rhd \mathrm{T}$ or $u\widehat{n} \rhd \mathrm{F}$. A number n *satisfies* a test u if $u\widehat{n} \rhd \mathrm{T}$. A test u is *satisfiable* if it is satisfied by some number.

Theorem 31. (Choose) *There is a procedure* C *such that for every test* u*:*

1. *If* u *is satisfiable, then* $\mathrm{C}u \rhd \widehat{n}$ *for some* n *satisfying* u.
2. *If* $\mathrm{C}u$ *evaluates, then* u *is satisfiable.*

Proof. We start with an auxiliary procedure H satisfying the recursive specification

$$H\,\widehat{n}\,u \;\equiv\; u\,\widehat{n}\,(\lambda\widehat{n})\,(\lambda(H(\mathrm{Succ}\ \widehat{n})u))\ \mathrm{I}$$

and define $\mathrm{C} := \lambda x.H\,\widehat{0}\,x$. Speaking informally, H realises a loop incrementing n until $u\,\widehat{n}$ succeeds. We say that $H\,\widehat{n}\,u$ is *ok* if $H\,\widehat{n}\,u \rhd \widehat{k}$ for some number k satisfying u and proceed as follows:

1. If n satisfies u, then $H\,\widehat{n}\,u$ is ok.

2. If $H\,\widehat{Sn}\,u$ is ok, then $H\,\widehat{n}\,u$ is ok.
3. If $H\,\widehat{n}\,u$ is ok, then $H\,\widehat{0}\,u$ is ok. Follows by induction on n with (2).
4. Claim 1 follows with (1) and (3).
5. If $H\,\widehat{n}\,u$ evaluates in k steps, then u is satisfiable. Follows by complete induction on k using the triangle property.
6. Claim 2 follows from (5) with $n = 0$. □

□

Note that the verification of choose employs in (6) complete induction on the step-index of an evaluation together with the triangle property (Fact 10) to handle the unguarded recursion of the auxiliary procedure H. This is the only time these devices are used in our development.

11 Results Obtained with Self-Interpreters

For the specification of a step-indexed self-interpreter, we define an injective encoding function for term options:

$$\begin{aligned}\widehat{\lfloor s \rfloor} &:= \lambda ab.a\overline{s}\\ \widehat{\emptyset} &:= \lambda ab.b\end{aligned}$$

Fact 32. *There is a procedure* E *such that* $\mathrm{E}\,\widehat{n}\,\overline{s} \equiv \widehat{eval\ n}\ s$ *for all* n *and* s.

Proof. We first construct and verify procedures realising the functions $m{=}n$ and s_u^k. We then construct and verify the procedure E following the recursive specification of the function *eval* in Section 9. □

Theorem 33. (Step-Indexed Self-Interpreter)

1. *If* $\mathrm{E}\,\widehat{n}\,\overline{s} \rhd \widehat{\lfloor t \rfloor}$, *then* $\mathrm{E}\,\widehat{Sn}\ \overline{s} \rhd \widehat{\lfloor t \rfloor}$.
2. $\forall sn.\ (\mathrm{E}\,\widehat{n}\,\overline{s} \rhd \widehat{\emptyset}) \vee (\exists t.\ \mathrm{E}\,\widehat{n}\,\overline{s} \rhd \widehat{\lfloor t \rfloor} \wedge s \rhd t)$.
3. *If* $s \rhd t$, *then* $\mathrm{E}\,\widehat{n}\,\overline{s} \rhd \widehat{\lfloor t \rfloor}$ *for some* n.

Proof. Follows with Facts 32 and 26. □

Theorem 34. (Totality) *The class of total procedures is not recognisable.*

Proof. By the reduction lemma. We define fs as a procedure that for closed s is total iff $\neg\mathcal{E}(s\overline{s})$. We define fs such that $(fs)\overline{t}$ evaluates if t is an application or an abstraction. If t is a number n, we evaluate $s\overline{s}$ with the step-indexed self-interpreter for n steps. If this succeeds, we diverge using D, otherwise we return I. Here are the definitions of f and the realising procedure v:

$$\begin{aligned}f &:= \lambda s.\lambda y.y\,(\lambda z.\mathrm{E}\,z\,(\overline{s\overline{s}})\,\mathrm{D}\,\mathrm{I})\,\mathrm{F}\,\mathrm{I}\\ v &:= \lambda x.\mathrm{L}(\mathrm{A}(\mathrm{A}(\mathrm{A}\,\overline{0}\,(\mathrm{L}(\mathrm{A}(\mathrm{A}(\mathrm{A}\,\overline{\mathrm{E}}\,\overline{0})(\mathrm{Q}\,(\mathrm{A}\,x(\mathrm{Q}\,x))))\,\overline{\mathrm{D}})\,\overline{\mathrm{I}}))\,\overline{\mathrm{F}})\,\overline{\mathrm{I}})\end{aligned}$$ □

□

Corollary 35. *The class of total terms is neither recognisable nor corecognisable.*

Proof. Suppose the class of total terms is recognisable. Then the class of total procedures is recognisable since the class of procedures is recognisable (follows with Fact 20). Contradiction with Theorem 34. The other direction is provided by Corollary 23. □

We now construct an unbounded self-interpreter using the procedure choose and the step-indexed self-interpreter E.

Theorem 36. (Self-Interpreter) *There is a procedure* U *such that:*

1. *If* $s \rhd t$*, then* $\mathrm{U}\,\overline{s} \rhd \overline{t}$*.*
2. *If* $\mathrm{U}\,\overline{s}$ *evaluates, then* s *evaluates.*

Proof. $\mathrm{U} := \lambda x.\, \mathrm{E}\,(\mathrm{C}(\lambda y.\mathrm{E}yx(\lambda\mathrm{T})\mathrm{F}))\,x\,\mathrm{I}\,\mathrm{I}$ does the job. The verification uses Theorems 33 and 31. □

Corollary 37. *The self-interpreter* U *recognises the class of evaluable terms.*

For Post's theorem we need a special self-interpreter considering two terms. We speak of a parallel or operator.

Theorem 38. (Parallel Or) *There is a procedure* O *such that:*

1. *If* s *or* t *evaluates, then* $\mathrm{O}\,\overline{s}\,\overline{t}$ *evaluates.*
2. *If* $\mathrm{O}\,\overline{s}\,\overline{t}$ *evaluates, then either* $\mathcal{E}\,s$ *and* $\mathrm{O}\,\overline{s}\,\overline{t} \rhd \mathrm{T}$*, or* $\mathcal{E}\,t$ *and* $\mathrm{O}\,\overline{s}\,\overline{t} \rhd \mathrm{F}$*.*

Proof. $\mathrm{O} := \lambda xy.\ (\lambda z.\mathrm{E}zx(\lambda\mathrm{T})(Ezy(\lambda\mathrm{F})\,\mathrm{I}))\ (\mathrm{C}(\lambda z.\mathrm{E}zx(\lambda\mathrm{T})(Ezy(\lambda\mathrm{T})\,\mathrm{F})))$ does the job. The verification uses Theorems 33 and 31. □

Corollary 39. (Post) *If* u *recognises* p *and* v *recognises* $\lambda s.\neg ps$*, then the procedure* $\lambda x.\mathrm{O}\,(\mathrm{A}\,\overline{u}\,(\mathrm{Q}x))\,(\mathrm{A}\,\overline{v}\,(\mathrm{Q}x))$ *decides* p *provided* p *is logically decidable.*

With parallel or we can also show that the family of recognisable classes is closed under union.

Corollary 40. (Union) *If* u *recognises* p *and* v *recognises* q*, then the procedure* $\lambda x.\mathrm{O}\,(\mathrm{A}\,\overline{u}\,(\mathrm{Q}x))\,(\mathrm{A}\,\overline{v}\,(\mathrm{Q}x))$ *recognises* $\lambda s.\ ps \vee qs$*.*

12 Enumerable Classes

A class is *enumerable* if there is a procedurally realisable function from numbers to term options that yields exactly the terms of the class. More precisely, a procedure u *enumerates* a class p if:

1. $\forall n.\ (\widehat{un} \rhd \widehat{\emptyset}) \vee (\exists s.\ \widehat{un} \rhd \widehat{\lfloor s \rfloor} \wedge ps)$.
2. $\forall s.\ ps \to \exists n.\ \widehat{un} \rhd \widehat{\lfloor s \rfloor}$.

Following well-known ideas, we show that a class is recognisable if and only if it is enumerable. We will be content with informal outlines of the proof in the Coq development since we have already seen all necessary formal techniques.

Fact 41. *Given an enumerator for p, one can construct a recogniser for p.*

Proof. Given a term s, the recogniser for p searches for a number n such that the enumerator for p yields s (using the procedure choose). □

Fact 42. *The class of a all terms is enumerable.*

Proof. One first writes an enumerator function and then translates it into a procedure. The translation to a procedure is routine. Coming up with a compact enumeration function is a nice programming exercise. Our solution is in the Coq development. □

Fact 43. *Given a recogniser for p, one can construct an enumerator for p.*

Proof. Given n, the enumerator for p obtains the term option for n using the term enumerator. If the option is not of the form $\lfloor ns \rfloor$, the enumerator for p fails. If the option is of the form $\lfloor ns \rfloor$, the recogniser for p is run on $\overline{s}$ for n steps using the step-indexed self-interpreter. If this succeeds, the enumerator for p succeeds with s, otherwise it fails. □

13 Markov's Principle

Markov's principle is a proposition not provable constructively and weaker than excluded middle [7]. Formulated for L, Markov's principle says that a class is decidable if it is recognisable and corecognisable. We establish two further characterisations of Markov's principle for L using parallel or (Theorem 38) and the enumerability of terms (Fact 42).

Lemma 44. *If p is decidable, then $\lambda\_.\exists s.ps$ is recognisable.*

Proof. Follows with Fact 42 and 31. □

Theorem 45. (Markov's Principle) *The following statements are equivalent:*

1. *If a class is recognisable and corecognisable, then it is decidable.*
2. *Satisfiability of decidable classes is stable under double negation:*
   $\forall p.\ \mathit{decidable}\ p \to \neg\neg(\exists s.ps) \to \exists s.ps.$
3. *Evaluation of closed terms is stable under double negation:*
   $\forall s.\ \mathit{closed}\ s \to \neg\neg\mathcal{E}s \to \mathcal{E}s.$

Proof. $1 \to 2$. Let p be decidable and $\neg\neg\exists s.ps$. We show $\exists s.ps$. By (1), Lemma 44, and $\neg(\exists s.ps) \leftrightarrow \bot$ we know that the class $\lambda\_.\exists s.ps$ is decidable. Thus we have either $\exists s.ps$ or $\neg\exists s.ps$. The first case is the claim and the second case is contradictory with the assumption.

$2 \to 3$. Let s be a closed term such that $\neg\neg\mathcal{E}s$. We show $\mathcal{E}s$. Consider the decidable class $p := \{\, n \mid \mathit{eval}\ n\ s \neq \emptyset \,\}$. We have $\mathcal{E}s \leftrightarrow \exists t.pt$. By (2) it suffices to show $\neg\neg\exists t.pt$, which follows with the assumption $\neg\neg\mathcal{E}s$.

$3 \to 1$. Let u be a recogniser for p and v be a recogniser for $\lambda s.\neg ps$. We show that $\lambda x.\mathrm{O}\,(\mathrm{A}\,\overline{u}\,(\mathrm{Q}x))\,(\mathrm{A}\,\overline{v}\,(\mathrm{Q}x))$ is a decider for p. By Theorem 38 it suffices to show that $\mathrm{O}\,(\overline{u\overline{s}})\,(\overline{v\overline{s}})$ evaluates for all terms s. Using (3) we prove this claim by contradiction. Suppose $\mathrm{O}\,(\overline{u\overline{s}})\,(\overline{v\overline{s}})$ does not evaluate. Then, using Theorem 38, neither $u\overline{s}$ nor $v\overline{s}$ evaluates. Thus $\neg ps$ and $\neg\neg ps$. Contradiction. □

We remark that Markov's principle for L follows from a global Markov's principle saying that satisfiability of functionally decidable classes of numbers is stable under double negation. This can be shown with Theorem 45 (3) and the equivalence $\mathcal{E}s \leftrightarrow \exists n.\, \mathit{eval}\, n\, s \neq \emptyset$.

References

1. A. Asperti and W. Ricciotti. Formalizing Turing machines. In *Logic, Language, Information and Computation*, pages 1–25. Springer, 2012.
2. A. Asperti and W. Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, Oct. 2015.
3. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.
4. A. Bauer. First steps in synthetic computability theory. *ENTCS*, 155:5–31, 2006.
5. G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 5th edition, 2007.
6. A. Ciaffaglione. Towards Turing computability via coinduction. *Science of Computer Programming*, 126:31–51, Sept. 2016.
7. T. Coquand and B. Mannaa. The independence of Markov's principle in type theory. In *FSCD 2016*, volume 52 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl, 2016.
8. U. Dal Lago and S. Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.
9. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2013.
10. J. M. Jansen. Programming in the λ-calculus: From Church to Scott and back. In *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013.
11. D. Kozen. *Automata and computability*. Springer, 1997.
12. T. Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.
13. J. Niehren. Functional computation as concurrent computation. In *POPL 1996*, pages 333–343. ACM, 1996.
14. M. Norrish. Mechanised computability theory. In *ITP 2011*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011.
15. The Coq Proof Assistant. `http://coq.inria.fr`.
16. J. Xu, X. Zhang, and C. Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In *ITP 2013*, volume 7998 of *LNCS*, pages 147–162. Springer, 2013.