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

Yannick Forster and Gert Smolka

Received: 26 February 2018 / Accepted: 4 October 2018

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.

Keywords Computability Theory · Lambda calculus · Coq · Type theory

1 Introduction

We study a minimal functional programming language L realising a subsystem of the λ-calculus [5] known as (weak) call-by-value λ-calculus [22,10]. As in most programming languages, β-reduction in 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 [14,16] 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 [25] 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

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

This is a pre-print of an article published in the Journal of Automated Reasoning. The final authenticated version is available online at: https://doi.org/10.1007/s10817-018-9484-2

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 as 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 [20, 10] 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 [18, 15], and program verification can be based on one-step reduction, the accompanying equivalence closure, and the connecting Church-Rosser property.

One place where the commitment to a constructive type 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 a special case of Markov's principle. Markov's principle does not hold in a purely constructive setting [9]. 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.

Related Work. This paper is an extended version of a previous conference paper [11]. Otherwise, there is not much work on computability theory in constructive type theory. We are aware of Asperti and Ricciotti [3, 4] 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 [8] formalises Turing machines coinductively in Coq and shows the agreement between a big-step and a small-step semantics.

Bauer [6] 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 [21] 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 [21] apart from the fact that Norrish works in a classical setting. Following Barendregt [5], Norrish works with full λ-calculus and Gödel-Church encodings. We work with call-by-value λ-calculus and Scott encodings instead. We remark that Gödel-Church encodings are not possible in a setting with weak β-reduction, and that Scott encodings are a simpler alternative to Gödel-Church encodings in full λ-calculus (since they don't involve recursion). Norrish proves Rice's theorem for partial recursive functions while we prove the theorem for procedures in L.

Xu, Zhang, and Urban [27] formalise Turing machines, abacus machines, and partial recursive functions in Isabelle (classical higher-order logic) and show their computational equivalence following Boolos et al. [7]. They prove the existence of a universal function. They do not consider the theorems of Rice and Post.

Dal Lago and Martini [10] 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 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.

This paper extends a previous conference paper [11]. Besides more detailed proofs, we have added Section 15 on computable functions and reducibility and Section 16 discussing future work. We are grateful for the comments of the anonymous reviewers, which helped to improve the presentation of the paper.

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`.

Organisation of the Paper. We start by defining the functional language L we will work with. We complement the defining big-step semantics with a uniformly confluent small-step semantics providing the base for the many program verifications in this paper. With the semantic tools in place, we define the Scott encodings of numbers and terms needed for the main results. We then prove the theorems of Scott and Rice. Next we extend our semantic tools with a step-indexed interpreter and prove that procedural decidability implies functional decidability. We then establish a procedure choose searching for a number satisfying a test. Choose will be the only procedure in the development using truly unguarded recursion. We then realise the step-indexed interpreter for L with a procedure in L and obtain several key results including unrecognisability of the class of total procedures, Post's theorem, and closure under union for recognisable classes. Next, we give translations between recognisers and enumerators, again exploiting the step-indexed self-interpreter. We then establish three characterisations of Markov's principle for L. Finally, we formalise many-one reductions and prove their basic properties. We conclude with a discussion of directions for future work.

2 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 $s \rhd t$ 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 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*:

$$s, t, u, v : \mathsf{T} \ ::= \ n \mid st \mid \lambda s \qquad (n : \mathsf{N})$$

We fix some terms for further use:

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

For readability, we will usually write concrete terms with named abstractions, as shown above. The Coq development provides a function translating terms with named abstractions to terms with de Bruijn indices.

Note that the term D defined above is reducible in the full λ-calculus but will not be reducible in L.

Note that the type T of terms is discrete (i.e., has functionally decidable equality). We will use this fact tacitly in proofs.

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 will be defined as closed abstractions and we will often exploit that substitutions do not affect closed terms.

We define a recursive boolean function $\mathit{bound}\, k\, s$ satisfying the equations

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

Speaking informally, $\mathit{bound}\, k\, s$ tests whether every free variable in s is smaller than k. We say that s is *bounded* by n if $\mathit{bound}\, n\, s = \mathsf{true}$. We now define *closed*

terms as terms bounded 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 1 *If s is bound by n and $k \geq n$, then $s_u^k = s$. Moreover, $s_u^k = s$ for closed s.*

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

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

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

Fact 2 T *and* F *are procedures such that* $\mathrm{T} \neq \mathrm{F}$ *and* $\mathrm{T}st \rhd s$ *and* $\mathrm{F}st \rhd t$ *for all procedures s, t.*

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

3 Reduction Semantics

To enable 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:

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

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 4 *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 $s \rhd t$.

The call-by-value λ-calculus in general and L in particular enjoy a strong confluence property [20,10] we call uniform confluence. Proving uniform confluence for L is easier than proving confluence for the full λ-calculus.

Fact 5 (Uniform Confluence) *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.*

Proof By induction on $s \succ t_1$.

1. Let $s = (\lambda s_1)(\lambda s_2)$. Then $t_1 = t_2$ since abstractions are irreducible.
2. Let $s = s_1 s_2$, $s_1 \succ s_1'$, and $t_1 = s_1' s_2$. Case analysis on $s \succ t_2$.
   (a) $s_1 \succ s_1''$ and $t_2 = s_1'' s_2$. The claim follows with the inductive hypothesis for $s_1 \succ s_1'$.
   (b) $s_2 \succ s_2'$, and $t_2 = s_1 s_2'$. The claim follows with $u = s_1' s_2'$.
3. Analogous to (2).

□

The intuitive reason why L is uniformly confluent is that only outermost β-redexes can be reduced and the reduction of a β-redex is functional. Hence, if $s \succ t_1$ and $s \succ t_2$, either the same redex is reduced and thus $t_1 = t_2$, or two disjoint redexes u_1 and u_2 are reduced and thus t_1 and t_2 can be joined by reducing the remaining redex u_2 in t_1 and the remaining redex u_1 in t_2.

As can be shown generally for abstraction reduction systems, uniform confluence has the important consequence that every weakly normalising term is strongly normalising. Moreover, uniform confluence implies parametric confluence, confluence and a property we call unique step index:

Fact 6 (Parametric Confluence) *If $s \succ^m t_1$ and $s \succ^n t_2$, then there exist numbers $k \leq n$ and $l \leq m$ and a term u such that $t_1 \succ^k u$ and $t_2 \succ^l u$ and $m + k = n + l$.*

Corollary 7 *$s \succ t$ is confluent.*

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

Corollary 8 (Unique Step Index) *If $s \rhd^m t$ and $s \rhd^n t$, then $m = n$.*

Corollary 9 (Triangle) *If $s \rhd^n t$ and $s \succ^+ s'$, then $s' \rhd^k t$ for some $k < n$.*

The triangle property does not hold for the full λ-calculus. In our setting, the triangle property is crucial to carry out partial correctness proofs for procedures employing unguarded recursion. The verification of a self-interpreter (Theorem 37) and a parallel-or operator (Theorem 39) are forthcoming examples of such proofs, which are done by complete induction on the step index n in $s \rhd^n t$. Given $s \rhd^n t$ and $s \succ^+ s'$, the triangle property gives us $s' \rhd^k t$ with $k < n$ so that the inductive hypothesis may apply to $s' \rhd^k t$.

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. Plotkin [22] defines an equivalence for a similar calculus which also applies below binders.

Fact 10 (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 7. Claim 3 follows with Claim 2, Fact 4 (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 $\mathrm{F}(\lambda s)(\lambda t)\mathrm{I} \succ^* t$.

4 Recursion and 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 [18, 15]. The Scott encoding of a value is a higher-order procedure providing the match construct for the value.

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

In contrast to Church encoding, Scott encoding does not directly enable structurally recursive definitions. We can however specify procedural recursion as follows:

Fact 11 (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 a *recursion operator* since it provides for recursive programming in L using well-known techniques from functional programming.

It is possible to define a procedure R in L such that Ru evaluates and $Ruv \equiv u(Ru)v$ for procedures u and v. Using this approach, recursive definitions Ru are no procedures and thus can not be passed as argument to higher-order functions directly, making verification difficult.

In the full λ-calculus, there is a procedure Y fulfilling $Yu \equiv u(Yu)$. For Y to serve as a recursion procedure in L, Yu would have to reduce to an abstraction in order to be passed as an argument to u, which is impossible for e.g. $u = \lambda x.\Omega$.

Programming with Scott encodings is convenient in that we can follow familiar patterns from functional programming. The translation of Coq functions to procedures is mechanical. The idea is that matches on inductive values translate to applications of the values, and that recursive functions are realized with the recursion operator.

We demonstrate the case with a *functional specification* of a procedure Add for addition:

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

We say that Add is a procedure *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):

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

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 suggests a recursive definition of Add using L's recursion operator ρ:

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

Note that the variable x is used to make recursive calls and that the body of the abstraction performs a match on the natural number value of the variable y.

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., $\mathrm{Add}\ \widehat{m}\ \widehat{n} \equiv \mathrm{Add}\ \widehat{n}\ \widehat{m}$) 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.

5 Scott Encoding of Terms

For the use of L as a model of computation it is essential that procedures can be analysed as data by procedures. We provide for this need with a Scott encoding of the inductive data type of terms. We define a *term encoding function* $\overline{s}$ 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.

One can show that the term encoding function is injective but we will not make explicit use of this fact in our proofs.

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 will be used in the proof of Rice's theorem and provides a functionality that is know as *quote* in the Scheme programming language.

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* $\mathrm{Q}\,\overline{s} \equiv \overline{\overline{s}}$.

6 Decidable and Recognisable Classes

Definition 13 *A* class (of terms) *is a unary predicate on terms. The letters p and q will range over classes of terms.* Deciders, recognizers, *and* corecognizers *of classes are procedures defined as follows:*

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

We say that a class is decidable *[*recognisable, corecognisable*] if it has a decider [recogniser, corecogniser].*

We establish the existence of undecidable and unrecognisable classes.

Fact 14 *Let u decide p. Then $ps \leftrightarrow u\overline{s} \rhd \mathrm{T}$ and $\neg ps \leftrightarrow u\overline{s} \rhd \mathrm{F}$.*

Fact 15 *$\lambda s.\neg(s\overline{s} \rhd \mathrm{T})$ is not decidable, and $\lambda s.\neg\mathcal{E}(s\overline{s})$ 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 will prove the following basic facts about decidable and recognisable classes of terms: 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 16 *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 17 $\lambda x.\mathrm{F}(ux)(vx)$ *recognises* $\lambda s.ps \wedge qs$ *if u recognises p and v recognises q.*

7 Scott's Theorem

We now prove Scott's theorem for L following Barendregt's proof [5] of Scott's theorem for the full λ-calculus. Scott's theorem says that every nontrivial class of closed terms that is closed under reduction equivalence is undecidable.

Fact 18 *Let s be closed. Then there exists a closed term t such that $t \equiv s\bar{t}$.*

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

Theorem 19 (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 18 gives us a closed term t such that $t \equiv v\bar{t} \equiv u\bar{t}(\lambda s_2)(\lambda s_1)\mathrm{I}$. Since u is a decider for p, we have two cases: (1) If $u\bar{t} \equiv \mathrm{T}$ and pt, then $t \equiv s_2$ contradicting $\neg ps_2$; (2) If $u\bar{t} \equiv \mathrm{F}$ and $\neg pt$, then $t \equiv s_1$ contradicting ps_1. □

Corollary 20 *The class of evaluating terms is undecidable.*

Corollary 21 *The class $\lambda s.s \equiv t$ is undecidable for every closed term t.*

8 Reduction Lemma

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

Fact 22 *The class* $\lambda s.\ closed\ s \land \neg\mathcal{E}(s\overline{s})$ *is not recognisable.*

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

Fact 23 *There is a decider for the class of closed terms.*

Proof The decider can be obtained with a procedure realising the boolean function $bound\, k\, s$ defined in Section 2. 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 4. □

Lemma 24 (Reduction) *A class p is unrecognisable if there exists a function $f : \mathsf{T} \to \mathsf{T}$ 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 23). 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 $\mathcal{E}(w\overline{s}) \leftrightarrow closed\ s \land \mathcal{E}(u(\overline{fs}))$. 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 $closed\ s \land \mathcal{E}(u(\overline{fs})) \leftrightarrow closed\ s \land \neg\mathcal{E}(s\overline{s})$ for all s. Thus w is recogniser for the unrecognisable class of Fact 22. Contradiction. □

9 Rice's Theorem

We now come to Rice's theorem. Rice's theorem says that every nontrivial class of procedures that is closed under semantic equivalence is undecidable. 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 will refer to this lemma as Rice's lemma (although we did not find it in the literature).

Semantic equivalence of terms is defined as follows:

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

Semantic equivalent terms have the same input output behaviour. If $s \approx t$ and $s\overline{u}$ evaluates to v, so does $t\overline{u}$, and if $s\overline{u}$ does not evaluate, neither does $t\overline{u}$.

We have $s \equiv t \to s \approx t$ using Fact 10. 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 25 (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. □

A term s is *total* if the application $s\overline{t}$ evaluates for every term t. If s and t are semantically equivalent terms, then s is total iff t is total.

Corollary 26 *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 27 (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 28 *The class of total procedures is undecidable.*

Rice's theorem looks similar to Scott's theorem but neither can be obtained from the other. Note 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, which builds a procedure that has the right properties. In textbooks (e.g., [14,16]), 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. In contrast, we have just seen a concise formal proof of Rice's theorem for procedures in L. The tools needed for meta-programming are simply the Scott encoding of terms and the accompanying procedures A, L, and Q.

10 Step-Indexed Interpreter and Modesty

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 have seen, L comes with unbounded recursion and thus admits nonterminating procedures. In contrast, Coq's type theory is designed such that functions always terminate.

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

- *logically decidable* if there is a proof of $\forall s.\ ps \vee \neg ps$.
- *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 will now prove that procedural decidability implies functional decidability, a property we call *modesty*. 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 $\mathsf{T}_\emptyset$ to denote the option type for T (the type of terms), and $\lfloor s \rfloor$ and $\emptyset$ to denote the values of $\mathsf{T}_\emptyset$. We define a function $\mathit{eval} : \mathsf{N} \to \mathsf{T} \to \mathsf{T}_\emptyset$ satisfying the following recursive specification.

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

Fact 29 *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 $\rightarrow$, follows by induction on $s \succ^* t$ and Claim 2. Claim 3, direction $\leftarrow$, follows by induction on n. □

Lemma 30 *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 29. Since the predicate $\lambda n.\ \exists t.\ \mathit{eval}\ n\ s = \lfloor t \rfloor$ is functionally decidable, constructive choice for 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 29. □

Theorem 31 (Modesty) *Procedurally decidable classes are functionally decidable.*

Proof Let u be a decider for p. Let s be a term. Lemma 30 gives us a term v such that $u\bar{s} \rhd v$. Now we return true if $v = \mathrm{T}$ and false otherwise. □

11 Choose

We give a procedure C that given a decidable test searches for a number satisfying the test. This is reminiscent of minimisation for recursive functions [7]. The procedure C will be the only procedure in our development using truly unguarded recursion. We will use C 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 32 (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 C employs for (6) complete induction on the step-index of an evaluation together with the triangle property (Fact 9) to handle the unguarded recursion of the auxiliary procedure H. This is the only time these devices are used in our development.

12 Results Obtained with Self-Interpreters

We now come to some of the key results of this paper:

- *Totality.* The class of total procedures is unrecognisable.
- *Self-interpreter.* There is a procedure U such that for all terms s, t:
  1. If $s \rhd t$, then $\mathrm{U}\,\overline{s} \rhd \overline{t}$.
  2. If $\mathrm{U}\,\overline{s}$ evaluates, then s evaluates.
- *Parallel or.* There is 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 $\mathrm{O}\,\overline{s}\,\overline{t} \rhd \mathrm{T}$ and $\mathcal{E}\,s$, or $\mathrm{O}\,\overline{s}\,\overline{t} \rhd \mathrm{F}$ and $\mathcal{E}\,t$.
- *Post's Theorem.* A class is decidable if it is recognisable, corecognisable, and logically decidable.
- *Closure under union.* The union of recognisable languages is recognisable.

We start with the construction of a step-indexed self-interpreter. For the specification of this procedure, we define an injective encoding function for term options:

$$\widetilde{\lfloor s \rfloor} \;:=\; \lambda ab.a\overline{s}$$
$$\widetilde{\emptyset} \;:=\; \lambda ab.b$$

Fact 33 *There is a procedure* E *such that* $\mathrm{E}\,\widehat{n}\,\overline{s} \equiv \widetilde{eval\ n\ s}$ *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 10. □

Theorem 34 (Step-Indexed Self-Interpreter)
The procedure E *satisfies the following properties:*

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

Proof Follows with Facts 33 and 29. □

Theorem 35 (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 represents 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:

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

Corollary 36 *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 23). Contradiction with Theorem 35. The other case is provided by Corollary 26. □

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

Theorem 37 (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 We define $u := \lambda x.\lambda y.\mathrm{E}yx(\lambda\mathrm{T})\mathrm{F}$ and $\mathrm{U} := \lambda x.\,\mathrm{E}\,(\mathrm{C}(u\,x))\,x\,\mathrm{I}\,\mathrm{I}$. Observe that $u\,\overline{s}$ is a test and that n satisfies $u\,\overline{s}$ if and only if $eval\,n\,s = \lfloor t' \rfloor$ for some t'.

For (1), assume that $s \rhd t$. Then there is n s.t. $eval\,n\,s = \lfloor t \rfloor$ and n satisfies $u\,\overline{s}$. By Theorem 32 (2) we get m satisfying $u\,\overline{s}$ with $C(u\,\overline{s}) \rhd \widehat{m}$. Thus $\mathrm{U}\,\overline{s} \equiv \mathrm{E}\,\widehat{m}\,\overline{s}\,\mathrm{I}\,\mathrm{I} \equiv \overline{t'}$ for some t' with $eval\,m\,s = \lfloor t' \rfloor$. By Lemmas 3 (2) and 29. we have $t = t'$ and $\mathrm{U}\,\overline{s} \rhd \overline{t}$.

For (2), assume that $\mathrm{U}\,\overline{s}$ evaluates. Then $C(u\,\overline{s})$ evaluates. By Theorem 32 (1) we get n and t with $eval\,n\,s = \lfloor t \rfloor$. Thus s evaluates by Lemma 29 (3). □

Corollary 38 *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 39 (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 is similar to the proof of Theorem 37. □

We can now also show that a class is decidable if it is recognisable, corecognisable, and logically decidable (Post's theorem).

Corollary 40 (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.*

Note that the assumption that p is logically decidable can be dropped if we assume excluded middle.

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

Corollary 41 (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 $\lambda s.\ ps \vee qs$.

13 Enumerable Classes

We now define enumerable classes and show that a class is enumerable if and only if it is recognisable.

Definition 42 *A procedure u* enumerates *a class p if:*

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

A procedure that enumerates a class is called an enumerator *for the class, and a class is called* enumerable *if it has an enumerator.*

Fact 43 *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). □

The translation of a recogniser to an enumerator is more involved and requires the step-indexed self-interpreter we already have (Theorem 34) and an enumerator for the class of all terms. The construction of the enumerator requires some effort. We first construct an enumeration function for the type of terms, which we then translate to a procedure enumerating the class of all terms.

Lemma 44 *Let $R : \mathsf{N} \to \mathsf{T}_\emptyset$ be a function and u be a procedure such that*

1. $\forall s \exists n.\ Rn = \lfloor s \rfloor$.
2. $\forall n.\ u\widehat{n} \rhd \widetilde{Rn}$.

Then u is an enumerator for the class of all terms.

Proof Straightforward. □

We construct the enumeration function for the type of terms using lists. We denote the inductive type of *lists over* X with $\mathcal{L}(X)$ and obtain lists with the constructors [] (nil) and $x :: A$ (cons). We write $A +\!\!+ B$ and $A[n] = \lfloor x \rfloor$ for applications of the recursive functions for *list concatenation* and *list position selection*.

We now define the function $R : \mathsf{N} \to \mathsf{T}_\emptyset$ required for Lemma 44 with a recursive auxiliary functions $L : \mathsf{N} \to \mathcal{L}(\mathsf{T})$ as follows:

$$\begin{aligned} L0 &= [\,] \\ L(Sn) &= Ln +\!\!+ n :: [st \mid s \in Ln, t \in Ln] +\!\!+ [\lambda s \mid s \in Ln] \\ Rn &:= (L(Sn))[n] \end{aligned}$$

Note the use of a notation for list comprehension, which abbreviates applications of a map function and a product function for lists.

Lemma 45

1. $|Ln| \geq n$.
2. $\forall n \exists s.\ (L(Sn))[n] = \lfloor s \rfloor$.
3. *If* $m \leq n$, *then* $Ln = Lm +\!\!+ A$ *for some* A.
4. *If* $A[k] = \lfloor s \rfloor$, *then* $(A +\!\!+ B)[k] = \lfloor s \rfloor$.
5. *If* $(Lm)[k] = \lfloor s \rfloor$ *and* $(Ln)[k] = \lfloor t \rfloor$, *then* $s = t$.

Proof Routine inductions. □

Lemma 46 $\forall s \exists n.\ Rn = \lfloor s \rfloor$.

Proof We define a recursive function $\beta : \mathsf{T} \to \mathsf{N}$ such that

$$\begin{aligned} \beta(n) &= Sn \\ \beta(st) &= S(\beta s + \beta t) \end{aligned}$$

and show by induction on s that $s \in L(\beta s)$ for all s.

Let s be a term. Since $s \in L(\beta s)$, we have $(L(\beta s))[n] = \lfloor s \rfloor$ for some n. By (2) and (5) of Lemma 45 we have $Rn = (L(Sn))[n] = \lfloor s \rfloor$. □

Fact 47 *The class of all terms is enumerable.*

Proof We translate the function R defined above and the necessary auxiliary functions into procedures using a Scott encoding for lists over terms. We relate each procedure with the specifying function with a canonical correctness lemma, as it was done before, for instance, for the step-indexed self-interpreter (Fact 33). With Lemmas 44 and 46 we then show that the procedure for R is an enumerator for the class of all terms. □

Fact 48 *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 $\lfloor ms \rfloor$, the enumerator for p fails. If the option is of the form $\lfloor ms \rfloor$, the recogniser for p is run on $\overline{s}$ for m steps using the step-indexed self-interpreter. If this succeeds, the enumerator for p succeeds with s, otherwise it fails. □

Theorem 49 *A class of terms is recognisable if and only if it is enumerable.*

Proof Facts 43 and 48. □

14 Markov's Principle

Markov's principle is a proposition not provable constructively [9] and weaker than excluded middle [13]. 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 39) and the enumerability of terms (Fact 47).

Lemma 50 *If p is decidable, then $\lambda\_.\exists s.ps$ is recognisable.*

Proof Follows with Fact 47 and 32. □

Theorem 51 (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 50, 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 39 it suffices to show that $\mathrm{O}\,(\overline{u\bar{s}})\,(\overline{v\bar{s}})$ evaluates for all terms s. Using (3), we prove this claim by contradiction. Suppose $\mathrm{O}\,(\overline{u\bar{s}})\,(\overline{v\bar{s}})$ does not evaluate. Then, using Theorem 39, neither $u\bar{s}$ nor $v\bar{s}$ evaluates. Thus $\neg ps$ and $\neg\neg ps$. Contradiction. □

We remark that Markov's principle for L follows from a global Markov's principle stating that satisfiability of functionally decidable classes of numbers is stable under double negation. This can be shown with Theorem 51 (3) and the equivalence $\mathcal{E}s \leftrightarrow \exists n.\ \mathit{eval}\,n\,s \neq \emptyset$.

15 Computable Functions and Reducibility

Another standard notion besides decidability and recognisability in computability theory is computability of functions. We say that a procedure u *computes* a function $f : \mathsf{T} \to \mathsf{T}$ if $\forall s.\ u\bar{s} \triangleright \overline{fs}$.

As for decidability, there is a modesty result for computable functions. Larchey-Wendling [17] proves a similar result, namely that all μ-recursively definable functions which are provably total in Coq correspond to functions in Coq.

For this we first need a decoding for the Scott encoding of terms.

Fact 52 (Decoding) *There is a function $\delta : \mathsf{T} \to \mathsf{T}_\emptyset$ such that (1) $\delta\,\bar{s} = \lfloor s \rfloor$ and (2) $\delta\,s = \lfloor t \rfloor \to \bar{t} = s$ for all terms s and t.*

Fact 53 (Modesty) *Let u be a procedure such that $\forall s \exists t.\ u\,\overline{s} \rhd \overline{t}$. Then there is a function $f : \mathsf{T} \to \mathsf{T}$ such that $\forall s.\ u\,\overline{s} \rhd \overline{fs}$.*

Proof Follows with Lemma 30 and Fact 52. □

In most of our unrecognisability proofs we did not prove unrecognisability directly, but instead deduced a contradiction by constructing a recogniser for e.g. $\lambda s.\ \neg\mathcal{E}(s\overline{s})$. Lemma 24 makes this technique explicit.

We now generalise this further by introducing the standard notion of (many-one) reducibility between classes. Intuitively, a class p is reducible to a class q if ps can be checked by checking qt for a computable witness t.

Formally, we define that a computable function $f : \mathsf{T} \to \mathsf{T}$ *reduces* a class p to a class q if $\forall s.\ ps \leftrightarrow q(fs)$. We then say that p *reduces to* q and write $p \preceq q$. Note that reducibility can also be formulated without referring to a function f. The advantage of the explicit function f is that the logical part of the reduction and its implementation can be clearly separated.

Fact 54 *$p \preceq q$ if and only if there is a procedure u s.t. $\forall s \exists t.\ u\overline{s} \rhd \overline{t} \wedge (ps \leftrightarrow qt)$.*

Proof For the direction from left to right, the procedure u computing the reduction f does the job. From right to left, we observe that $\forall s.\ \exists t. u\overline{s} \rhd t$ and use Fact 53 to obtain a function f computed by u. □

Fact 55 *The following hold:*

1. *$\preceq$ is reflexive.*
2. *$\preceq$ is transitive.*
3. *If $\forall s.\ ps \leftrightarrow p's$ and $\forall s.\ qs \leftrightarrow q's$ then $p \preceq q \to p' \preceq q'$.*
4. *$p \preceq \lambda s.\top$ if and only if $\forall s.\ ps$.*
5. *$p \preceq \lambda s.\bot$ if and only if $\forall s.\ \neg ps$.*

Proof (1) $fs := s$ does the job. (2) Let f reduce p to q and g reduce q to r. Then $p \preceq r$ via $hs := g(fs)$. (3) If $p \preceq q$ via f, then also $p' \preceq q'$ via f. (4,5) The direction from left to right is immediate, the direction from right to left follows from (3). □

Fact 56 *The following hold:*

1. *If q is decidable and $p \preceq q$, then p is decidable.*
2. *If q is recognisable and $p \preceq q$, then p is recognisable.*

Proof If u decides q, f reduces p to q and v computes f, $\lambda x.u(vx)$ decides p. The proof for (2) is similar. □

Corollary 57 *If p is undecidable (unrecognisable) and $p \preceq q$, then q is undecidable (unrecognisable).*

Note that Lemma 24 is an instance of this corollary for $p := \lambda s.\neg\mathcal{E}(s\overline{s})$.

We now show that $\mathcal{E}$ is a greatest element of the class of recognisable classes under $\preceq$. That $\mathcal{E}$ is recognisable follows from Corollary 38.

Fact 58 *A class p is recognisable if and only if $p \preceq \mathcal{E}$.*

Proof The direction from right to left follows from Fact 56 (2) and Corollary 38. For the other direction, let u recognise p. Then $fs := u\overline{s}$ reduces p to $\mathcal{E}$. □

We also prove that decidable classes are lower bounds for nontrivial classes under $\preceq$ and that any two classes have a least upper bound:

Fact 59 *If p is decidable and q is nontrivial, then $p \preceq q$.*

Proof Let g be a computable functional decider for p using Theorem 31, qt_1 and $\neg qt_2$. Then $fs := \mathbf{if}\ gs\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$ reduces p to q. □

Fact 60 *Two classes p and q always have a least upper bound w.r.t $\preceq$.*

Proof Let p and q be classes. We use the tree structure of terms and define:

$$rs := \begin{cases} ps' & \text{if } s = 0s' \\ qs' & \text{if } s = (Sn)s' \\ qs & \text{otherwise} \end{cases}$$

Now $\lambda s.\ 0s$ reduces p to r and $\lambda s.\ 1s$ reduces q to r.

Let r' be a class s.t. $p \preceq r'$ via f and $q \preceq r'$ via g. Then

$$hs := \begin{cases} fs' & \text{if } s = 0s' \\ gs' & \text{if } s = (Sn)s' \\ gs & \text{otherwise} \end{cases}$$

reduces r to r'. □

Classically p reduces to q if and only if the complement of p reduces to the complement of q. The direction from left to right holds constructively.

Fact 61 *If $p \preceq q$, then $(\lambda s.\ \neg ps) \preceq (\lambda s.\ \neg qs)$.*

Proof $ps \leftrightarrow q(fs)$ implies $\neg ps \leftrightarrow \neg q(fs)$. □

It seems unlikely that the other direction is provable constructively, as it implies Markov's principle for L:

Fact 62 *If $\forall pq.\ (\lambda s.\ \neg ps) \preceq (\lambda s.\ \neg qs) \to p \preceq q$, then Markov's principle for L holds.*

Proof We prove characterisation (3) from Theorem 51, i.e. that $\neg\neg\mathcal{E}s$ implies $\mathcal{E}s$. By easy logical reasoning, this holds if $\mathcal{E} \preceq (\lambda s.\ \neg\neg\mathcal{E}s)$. By assumption, it suffices to prove that $(\lambda s.\ \neg\mathcal{E}s) \preceq (\lambda s.\ \neg\neg\neg\mathcal{E}s)$, which holds by Fact 55 (3). □

16 Future work

Stack Machines. Plotkin [22] defines a stack machine for his call-by-value λ-calculus, which uses an evaluation order similar to the order used in our evaluation relation ($\triangleright$). His proofs, while explained very carefully and in detail, do not make use of modern tools like inductive predicates or structural induction.

We want to verify a stack machine for L, which also can be considered as a first step of simulating L using Turing machines.

Automated Extraction. It is believed that every program definable in Coq without axioms is computable. While this can not be proven inside the theory of

Coq, it is still possible to translate every individual Coq function to L and prove the correctness of the resulting procedure.

In fact, this translation can be mechanised. We are working on a tactics framework that allows the automatic verified translation of a large class of Coq functions into L. Using this framework, the Coq development of this article could be considerably shortened, because explicit verification of total procedures becomes unnecessary.

Undecidability Proofs. Most introductory books on computation also give undecidability proofs for problems which do not mention the model of computation explicitly, for instance for Post's correspondence problem [24].

We want to verify such reductions, including reductions concerning the undecidability of various logics, ultimately working towards a library of undecidable problems.

Reducibility Degrees and Turing Reducibility. Many-one reducibility as we defined it can be extended to a strict order on so called reducibility degrees, which are equivalence classes of the equivalence relation induced by $\preceq$.

An interesting question is whether there are recognisable classes strictly between decidable classes and $\mathcal{E}$. They would be undecidable classes whose complements fail to be recognisable, but where unrecognisability can not be proven by reduction from $\lambda s.\ \neg\mathcal{E}s$ or $\lambda s.\ \neg\mathcal{E}(s\bar{s})$, which was possible for all recognisable classes considered in this paper. Post [23] constructs a "simple" set which yields such a class.

We want to formalise the theory of reducibility degrees including the construction of a simple set. The theory can then be extended further to the notion of Turing reducibility, which needs a definition of L with oracles. For Turing reducibility, simple sets do not suffice to establish intermediate degrees, and their existence is proven via the more complicated priority method by Friedberg and Muchnik [12, 19].

Complexity Theory. Formalised complexity theory may be considered as a natural follow-up topic to formalised computability theory. To formalise complexity theory, we first have to formalise a model of computability with reasonable time and space measures. However, the only models with well-understood reasonable time and space measures are Turing machines and RAM machines [26]. Serious verified programming using those models, even to the extent as it was needed for this paper, would be unfeasible.

Dal Lago and Martini [10] give a reasonable time measure for their weak call-by-value λ-calculus by simulating it using Turing machines, but do not consider space. Accattoli and Dal Lago provide a polynomial algorithm to simulate full λ-calculus [2] and a polynomial implementation of Turing machines in the full λ-calculus [1]. We think that L is a sweet spot for complexity theory and want to extend the existing work to also include space, which would establish L as a model for complexity theory in Coq.

References

1. B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

2. B. Accattoli and U. Dal Lago. (Leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12, 2016.
3. A. Asperti and W. Ricciotti. Formalizing Turing machines. In *Logic, Language, Information and Computation*, pages 1–25. Springer, 2012.
4. A. Asperti and W. Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, Oct. 2015.
5. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.
6. A. Bauer. First steps in synthetic computability theory. *ENTCS*, 155:5–31, 2006.
7. G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 5th edition, 2007.
8. A. Ciaffaglione. Towards Turing computability via coinduction. *Science of Computer Programming*, 126:31–51, Sept. 2016.
9. 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.
10. U. Dal Lago and S. Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.
11. Y. Forster and G. Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017*, pages 189–206. Springer, LNCS 10499, 2017.
12. R. M. Friedberg. Two recursively enumerable sets of incomparable degrees of unsolvability (solution of post's problem, 1944). *Proceedings of the National Academy of Sciences*, 43(2):236–238, 1957.
13. H. Herbelin. An intuitionistic logic that proves Markov's principle. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 50–56, 2010.
14. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2013.
15. 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.
16. D. Kozen. *Automata and computability*. Springer, 1997.
17. D. Larchey-Wendling. Typing total recursive functions in coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 371–388, 2017.
18. T. Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.
19. A. A. Muchnik. On the unsolvability of the problem of reducibility in the theory of algorithms. In *Dokl. Akad. Nauk SSSR*, volume 108, page 1, 1956.
20. J. Niehren. Functional computation as concurrent computation. In *POPL 1996*, pages 333–343. ACM, 1996.
21. M. Norrish. Mechanised computability theory. In *ITP 2011*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011.
22. G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
23. E. L. Post. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society*, 50(5):284–316, 1944.
24. E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.
25. The Coq Proof Assistant. `http://coq.inria.fr`.
26. P. Van Emde Boas. Machine models and simulations. *Handbook of Theoretical Computer Science, volume A*, pages 1–66, 1991.
27. 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.