Oracle Computability and Turing Reducibility in the Calculus of Inductive Constructions*

Yannick Forster[1] [0000−0002−8676−9819] ✉, Dominik Kirst[2,3] [0000−0003−4126−6975], and Niklas Mück[3] [0009−0006−9622−0762]

[1] Inria, LS2N, Université Nantes, France
yannick.forster@inria.fr
[2] Ben-Gurion University of the Negev, Beer-Sheva, Israel
kirst@cs.bgu.ac.il
[3] Saarland University and MPI-SWS, Saarland Informatics Campus, Saarbrücken, Germany
mueck@mpi-sws.org

Abstract. We develop synthetic notions of oracle computability and Turing reducibility in the Calculus of Inductive Constructions (CIC), the constructive type theory underlying the Coq proof assistant. As usual in synthetic approaches, we employ a definition of oracle computations based on meta-level functions rather than object-level models of computation, relying on the fact that in constructive systems such as CIC all definable functions are computable by construction. Such an approach lends itself well to machine-checked proofs, which we carry out in Coq.

There is a tension in finding a good synthetic rendering of the higher-order notion of oracle computability. On the one hand, it has to be informative enough to prove central results, ensuring that all notions are faithfully captured. On the other hand, it has to be restricted enough to benefit from axioms for synthetic computability, which usually concern first-order objects. Drawing inspiration from a definition by Andrej Bauer based on continuous functions in the effective topos, we use a notion of sequential continuity to characterise valid oracle computations.

As main technical results, we show that Turing reducibility forms an upper semilattice, transports decidability, and is strictly more expressive than truth-table reducibility, and prove that whenever both a predicate p and its complement are semi-decidable relative to an oracle q, then p Turing-reduces to q.

Keywords: Type theory · Logical foundations · Synthetic computability theory · Coq proof assistant

* Yannick Forster received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101024493. Dominik Kirst is supported by a Minerva Fellowship of the Minerva Stiftung Gesellschaft fuer die Forschung mbH. The published version of this paper is available at XXXX.

1 Introduction

In recent years, synthetic computability theory [37,5,1,2] has gained increasing attention in the fields of constructive mathematics and interactive theorem proving [16,10,12,24,39,40]. In contrast to the usual analytic approach based on describing the functions considered computable by means of a model like Turing machines, μ-recursive functions, or the λ-calculus, the synthetic approach exploits that in a constructive setting no non-computable functions can be defined in the first place, making a later description of the computable fragment obsolete. This idea enables much more compact definitions and proofs, for instance decidability of sets over $\mathbb{N}$ can be expressed by equivalence to functions $f : \mathbb{N} \to \mathbb{B}$ without any further computability requirement regarding f, simplifying a formal mathematical development.

Furthermore, synthetic computability is the only approach to computability enabling a feasible mechanisation using a proof assistant. The general value of machine-checking important foundational results, for instance to obtain a library of mathematics and theoretical computer science, has become more appreciated in more and more subcommunities, up to the point that some mechanisations of results reach cutting edge research. However, even though machine-checked mathematics has a long history, computability theory, and even more so relative computability theory based on oracles, have not been tackled to a substantial amount past basic results such as Rice's theorem before the use of synthetic computability. This is because there is a big amount of "invisible" mathematics [4] that has to be made explicit in proof assistants, due to the use of the informal Church Turing thesis on paper that cannot be formally replicated. Filling in these missing details is infeasible, to the amount that textbook computability theory based on models of computations and the informal Church Turing thesis is not really formalisable to a reasonable extent.

The synthetic perspective remedies these issues and has been fruitfully used to describe basic concepts in computability theory in proof assistants. The approach is especially natural in constructive type theories such as the Calculus of Inductive Constructions (CIC) [6,34] underlying the Coq proof assistant [41]: as CIC embodies a dependently-typed functional programming language, every definable function conveys its own executable implementation.

However, the synthetic characterisation of oracle computations in general (i.e. algorithms relative to some potentially non-computable subroutine) and Turing reductions in particular (i.e. decision procedures relative to some oracle giving answer to a potentially non-decidable problem) has turned out to be more complicated. First, a Turing reduction cannot naively be described by a transformation of computable decision procedures $\mathbb{N} \to \mathbb{B}$ as this would rule out the intended application to oracles for problems that can be proved undecidable using usual axioms of synthetic computability such as Church's thesis (CT). Secondly, when instead characterising Turing reductions by transformations of possibly non-computable decision procedures represented as binary relations $\mathbb{N} \to \mathbb{B} \to \mathbb{P}$, one has to ensure that computability is preserved in the sense that computable oracles induce computable reductions in order to enable intended properties like the transport of (un-)decidability. Thirdly, to rule out exotic reductions whose behaviour on non-computable oracles differs substantially from their action on computable oracles, one needs to impose a form of continuity.

The possible formulations of continuity of functionals on partial spaces such as $\mathbb{N} \to \mathbb{B} \to \mathbb{P}$ are numerous: Bauer [3], who gave the first synthetic definition of oracle computability we draw our inspiration from, employs the order-theoretic variant of functionals preserving suprema in directed countable partial orders. The first author of this paper [11] describes a reformulation in CIC in joint work with the second author, using a modified variant of modulus continuity where every terminating oracle computation provides classical information about the information accessed from the oracle. We have suggested a more constructive formulation of modulus continuity in past work [15] and established Post's theorem connecting the arithmetical hierarchy with Turing degrees for this definition [22,30]. However, this proof assumes an enumeration of all (higher-order) oracle computations defined via modulus continuity, which seems not to follow from CT, therefore leaving the consistency status of the assumption unclear.

As a remedy to this situation, we propose an alternative synthetic characterisation of oracle computability based on a stricter notion of sequential continuity, loosely following van Oosten [33]. Concretely, a sequentially continuous function with input type I and output type O with an oracle expecting questions of type Q and giving answers of type A can be represented by a partial function $\tau : I \to A^* \rightharpoonup Q + O$, where τi can be seen as a (potentially infinite) tree. Concretely, τi is a function that maps paths of type A^* (i.e. edges are labeled by elements of type A) to inner nodes labeled by Q and leafs labeled by O.

While this concept naturally describes the functionals considered computable by emphasising the sequence of computation steps interleaved with oracle interactions, it immediately yields the desired enumeration from CT by reducing higher-order functionals on partial spaces to partial first-order functions.

In this paper we develop the theory of oracle computability as far as possible without any axioms for synthetic computability: we show that Turing reducibility forms an upper semilattice, transports decidability, and is strictly more expressive than truth-table reducibility, and prove that whenever both a predicate p and its complement are semi-decidable relative to an oracle q, then p Turing-reduces to q.[1] All results are mechanised in Coq, both to showcase the feasibility of the synthetic approach and as base for future related mechanisation projects.

For easy accessibility, the Coq development[2] is seamlessly integrated with the text presentation: every formal statement in the PDF version of this paper is hyperlinked with HTML documentation of the Coq code. To further improve fluid

[1] The non-relativised form of the latter statement also appears under the name of "Post's theorem" in the literature [42], not to be confused with the mentioned theorem regarding the arithmetical hierarchy, see the explanation in Section 9.

[2] `https://github.com/uds-psl/coq-synthetic-computability/tree/code-paper-oracle-computability`

readability, we introduce most concepts and notations in passing, but hyperlink most definitions in the PDF with the glossary in Appendix A.

Contribution We give a definition of synthetic oracle computability in constructive type theory and derive notions of Turing reducibility and relative semi-decidability. We establish basic properties of all notions, most notably that Turing reducibility forms an upper semi-lattice, transports decidability if and only if Markov's principle holds, and is strictly more general than truth-table reducibility. We conclude by a proof of Post's theorem relating decidability with semi-decidability of a set and its complement.

Outline We begin by introducing the central notion of synthetic oracle computability in Section 2, employed in Section 3 to derive synthetic notions of Turing reducibility and oracle semi-decidability. Before we discuss their respective properties (Sections 6 and 7) and show that Turing reducibility is strictly weaker than a previous synthetic rendering of truth-table reducibility (Section 8), we develop the basic theory of synthetic oracle computations by establishing their closure properties (Section 4) and by capturing their computational behaviour (Section 5). Some of these closure properties rely on a rather technical alternative characterisation of oracle computability described in Appendix B, which will also be used to establish the main result relating oracle semi-decidability with Turing reducibility discussed in Section 9. We conclude in Section 10 with remarks on the Coq formalisation as well as future and related work.

2 Synthetic Oracle Computability

The central notion of this paper is the synthetic definition of oracle computability. Historically, oracle computability was introduced as an extension of Turing machines in Turing's PhD thesis [43], but popularised by Post [35]. Various analytic definitions of oracle computability exist, all having in common that computations can ask questions and retrieve answers from an oracle.

For our synthetic definition, we specify concretely when a higher-order functional $F\colon (Q\to A\to\mathbb{P})\to(I\to O\to\mathbb{P})$ is considered (oracle-)computable. Such a functional takes as input a possibly non-total binary relation $R\colon Q\to A\to\mathbb{P}$, an oracle relating questions $q\colon Q$ to answers $a\colon A$, and yields a computation relating inputs $i\colon I$ to outputs $o\colon O$. For special cases like Turing reductions, we will instantiate $Q, I := \mathbb{N}$ and $A, O := \mathbb{B}$. Note that we do not require oracles R to be deterministic, but if they are, then so are the resulting relations FR (cf. Lemma 11).

We define oracle computability by observing that a terminating computation with oracles has a sequential form: in any step of the sequence, the oracle computation can ask a question to the oracle, return an output, or diverge. Informally, we can enforce such sequential behaviour by requiring that every terminating computation $FR\,i\,o$ can be described by (finite, possibly empty) lists $qs\colon Q^*$ and $as\colon A^*$ such that from the input i the output o is eventually obtained after a finite sequence of steps, during which the questions in qs are asked to the oracle one-by-one, yielding corresponding answers in as. This computational data can be captured by a partial[3] function of type $I \to A^* \rightharpoonup Q + O$, called the (computation) tree of F, that on some input and list of previous answers either returns the next question to the oracle, returns the final output, or diverges.

So more formally, we call $F\colon (Q \to A \to \mathbb{P}) \to (I \to O \to \mathbb{P})$ an (oracle-)computable functional if there is a tree $\tau\colon I \to A^* \rightharpoonup Q + O$ such that

$$\forall R\, i\, o.\; F R\, i\, o \;\leftrightarrow\; \exists qs\; as.\; \tau i\,;R \vdash qs\,;as \;\wedge\; \tau\, i\, as \triangleright \mathsf{out}\; o$$

with the interrogation relation $\sigma; R \vdash qs; as$ being defined inductively by

$$\frac{}{\sigma\,;R \vdash [\,]\,;[\,]} \qquad\qquad \frac{\sigma\,;R \vdash qs\,;as \qquad \sigma\, as \triangleright \mathsf{ask}\; q \qquad Rqa}{\sigma\,;R \vdash qs +\!\!+ [q]\,;as +\!\!+ [a]}$$

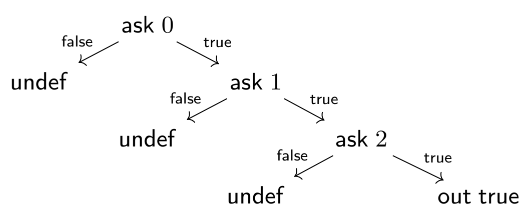
where A^* is the type of lists over a, $l +\!\!+ l'$ is list concatenation, where we use the suggestive shorthands $\mathsf{ask}\; q$ and $\mathsf{out}\; o$ for the respective injections into the sum type $Q + O$, and where $\sigma\colon A^* \rightharpoonup Q + O$ denotes a tree at a fixed input i.

To provide some further intuition and visualise the usage of the word "tree", we discuss the following example functional in more detail:

$$\begin{aligned} F &: (\mathbb{N} \to \mathbb{B} \to \mathbb{P}) \to (\mathbb{N} \to \mathbb{B} \to \mathbb{P}) \\ F R\, i\, o &:= o = \mathsf{true} \wedge \forall q < i.\, R\, q\, \mathsf{true} \end{aligned}$$

Intuitively, the functional can be computed by asking all questions q for $q < i$ to the oracle. If the oracle does not return any value, F does not return a value. If the oracle returns false somewhere, F also does not return a value – i.e. runs forever. If the oracle indeed returns true for all $q < i$, F returns true.

In the case of $i = 3$, this process may be depicted by



where the paths along labelled edges represent the possible answer lists as while the nodes represent the corresponding actions of the computation: the paths along inner nodes denote the question lists qs and the leafs the output behaviour. Note that $\mathsf{ret} : X \rightharpoonup X$ is the return of partial functions, turning a value into an always defined partial value, while undef denotes the diverging partial value. Formally, a tree $\tau\colon \mathbb{N} \to \mathbb{B}^* \rightharpoonup \mathbb{N} + \mathbb{B}$ computing F can be defined by

$$\tau\, i\, as \;:=\; \begin{cases} \mathsf{undef} & \text{if } \mathsf{false} \in as \\ \mathsf{ret}\;(\mathsf{ask}\;\; |as|) & \text{if } \mathsf{false} \notin as \wedge |as| < i \\ \mathsf{ret}\;(\mathsf{out}\;\; \mathsf{true}) & \text{if } \mathsf{false} \notin as \wedge |as| \geq i \end{cases}$$

[3] There are many ways how semi-decidable partial values can be represented in CIC, for instance via step-indexing. Since the actual implementation does not matter, we abstract over any representation providing the necessary operations, see Appendix A.

where here and later on we use such function definitions by cases to represent (computable) pattern matching.

As usual in synthetic mathematics, the definition of a functional F as being computable if it can be described by a tree is implicitly relying on the fact that all definable (partial) functions in CIC could also be shown computable in the analytic sense. Describing oracle computations via trees in stages goes back to Kleene [25], cf. also the book by Odifreddi [31]. Our definition can be seen as a more explicit form of sequential continuity due to van Oosten [32,33], or as a partial, extensional form of a dialogue tree due to Escardó [9]. Our definition allows us to re-prove the theorem by Kleene [26] and Davis [7] that computable functionals fulfill the more common definition of continuity with a modulus:

Lemma 1. *Let F be a computable functional. If $F R\, i\, o$, then there exists a list $qs : Q^*$, the so-called modulus of continuity, such that $\forall q \in qs.\ \exists a.\ Rqa$ and for all R' with $\forall q \in qs. \forall a.\ Rqa \leftrightarrow R'qa$ we also have that $FR'\, i\, o$.*

Proof. Given $F R\, i\, o$ and F computable by τ we have $\tau i\, ; R \vdash qs\, ; as$ and $\tau\, i\, as \triangleright \mathsf{out}\ o$. It suffices to prove both $\forall q \in qs.\ \exists a.\ Rqa$ and $\tau i\, ; R' \vdash qs\, ; as$ by induction on the given interrogation, which is trivial. □

Nevertheless, our notion of computable functionals is strictly stronger than modulus continuity as stated, while we are unaware of a proof relating it to a version where the moduli are computed by a partial function.

Lemma 2. *There are modulus-continuous functionals that are not computable.*

Proof. Consider the functional $F : (\mathbb{N} \to \mathbb{B} \to \mathbb{P}) \to (I \to O \to \mathbb{P})$ defined by

$$F Rio \ := \ \exists q.\, R\, q\, \mathsf{true}.$$

Clearly, F is modulus-continuous since from a terminating run $FRio$ we obtain q with $R\, q\, \mathsf{true}$ and therefore can choose $qs := [q]$ as suitable modulus.

However, suppose $\tau : I \to \mathbb{B}^* \rightharpoonup \mathbb{N} + O$ were a tree for F, then given some input i we can inspect the result of $\tau\, i\, []$ because $F\, R_\top\, i\, o$ holds for all i, o, and the full oracle $R_\top\, q\, a := \top$. However, the result cannot be $\mathsf{out}\ o$ for any output o, as this would yield $FR_\bot$ for the empty oracle $R_\bot\, q\, a := \bot$, violating the definition of F. Thus $\tau\, i\, [] \triangleright \mathsf{ask}\ q_0$, conveying an initial question q_0 independent of the input oracle. But then employing the oracle R_0 defined by $R_0\, q_0\, a := \bot$ and $R_0\, q\, a := \top$ for all $q \neq q_0$ we certainly have $F\, R_0\, i\, o$ by definition but no interrogation $\tau i\, ; R_0 \vdash qs\, ; as$ with $\tau\, i\, as \triangleright \mathsf{out}\ o$, as this would necessarily include an answer a with $R_0\, q_0\, a$ as first step, contradicting the construction of R_0. □

The advantage of using the stricter notion of sequential continuity over modulus continuity is that by their reduction to trees, computable functionals are effectively turned into flat first-order functions on data types. Thus one directly obtains an enumeration of all oracle computations, as needed in most advanced scenarios, from an enumeration of first-order functions, which itself could be obtained by assuming usual axioms for synthetic computability.

3 Turing Reducibility and Oracle Semi-Decidability

Using our synthetic notion of oracle computability, we can directly derive synthetic formulations of two further central notions of computability theory: Turing reducibility – capturing when a predicate is decidable relative to a given predicate – and oracle semi-decidability – capturing when a predicate can be recognised relative to a given predicate. To provide some intuition first, we recall that in the synthetic setting a predicate $p : X \to \mathbb{P}$ over some type X is decidable if there is a function $f\colon X\to\mathbb{B}$ such that $\forall x.\ px \leftrightarrow fx = \mathsf{true}$, i.e. f acts as a decider of p. This definition is standard in synthetic computability [1,16] and relies on the fact that constructively definable functions $f\colon X\to\mathbb{B}$ are computable.

To relativise the definition of a decider to an oracle, we first define the characteristic relation $\hat{p}\colon X\to\mathbb{B}\to\mathbb{P}$ of a predicate $p\colon X\to\mathbb{P}$ by

$$\hat{p} \ := \ \lambda xb. \begin{cases} px & \text{if } b = \mathsf{true} \\ \neg px & \text{if } b = \mathsf{false}. \end{cases}$$

Employing $\hat{p}$, we can now equivalently characterise a decider f for p by requiring that $\forall xb.\ \hat{p}xb \leftrightarrow fx = b$. Relativising this exact pattern, we then define Turing reducibility of a predicate $p\colon X\to\mathbb{P}$ to $q\colon Y\to\mathbb{P}$ by a computable functional F transporting the characteristic relation of q to the characteristic relation of p:

$$p \preceq_{\mathsf{T}} q \ := \ \exists F.\ F \textit{ is computable} \land \forall xb.\ \hat{p}xb \leftrightarrow F\hat{q}xb$$

Note that while we do not need to annotate a decider f with a computability condition because we consider all first-order functions of type $\mathbb{N}\to\mathbb{N}$ or $\mathbb{N}\to\mathbb{B}$ as computable, a Turing reduction is not first-order, and thus needs to be enriched with a tree to rule out unwanted behaviour. In fact, without this condition, we would obtain $p \preceq_{\mathsf{T}} q$ for every p and q by simply setting $F\,R := \hat{p}$.

Next, regarding semi-decidability, a possible non-relativised synthetic definition is to require a partial function $f\colon X\rightharpoonup\mathbb{1}$ such that $\forall x.\ px \leftrightarrow fx \triangleright \star$, where $\mathbb{1}$ is the inductive unit type with singular element $\star$. That is, the semi-decider f terminates on elements of p and diverges on the complement $\overline{p}$ of p (cf. [11]).

Again relativising the same pattern, we say that $p\colon X\to\mathbb{P}$ is (oracle-)semi-decidable relative to $q\colon Y\to\mathbb{P}$ if there is a computable functional F mapping relations $R\colon Y\to\mathbb{B}\to\mathbb{P}$ to relations of type $X\to\mathbb{1}\to\mathbb{P}$ such that $F\hat{q}$ accepts p:

$$\mathcal{S}_q(p) \ := \ \exists F.\ F \textit{ is computable} \land \forall x.\ px \leftrightarrow F\hat{q}x\star$$

As in the case of Turing reductions, the computability condition of an oracle semi-decider is crucial: without the restriction, we would obtain $\mathcal{S}_q(p)$ for every p and q by setting $F\,R\,x\,\star := p\,x$.

While we defer developing the theory of synthetic Turing reducibility and oracle semi-decidability to later sections, we can already record here that the fact that decidability implies semi-decidability also holds in relativised form:

Lemma 3. *If* $p \preceq_{\mathsf{T}} q$ *then* $\mathcal{S}_q(p)$ *and* $\mathcal{S}_q(\overline{p})$.

Proof. Let F witnesses $p \preceq_{\mathsf{T}} q$, then $F'\,R\,x\,\star := F\,R\,x\,\mathsf{true}$ witnesses $\mathcal{S}_q(p)$. In particular, if $\tau\colon X\to\mathbb{B}^*\rightharpoonup\mathbb{N}+\mathbb{B}$ computes F, then $\tau'\colon X\to\mathbb{B}^*\rightharpoonup\mathbb{N}+\mathbb{1}$, constructed by running τ and returning $\mathsf{out}\ \star$ whenever τ returns $\mathsf{out}\ \mathsf{true}$, computes F'. The proof of $\mathcal{S}_q(\overline{p})$ is analogous, simply using false in place of true. □

4 Closure Properties of Oracle Computations

In this section we collect some examples of computable functionals and show how they can be composed, yielding a helpful abstraction for later computability proofs without need for constructing concrete computation trees. Note that the last statements of this section depend on a rather technical intermediate construction using a more flexible form of interrogations. We refer to the Coq code and to Appendix B, where we will also deliver the proofs left out.

First, we show that composition with a transformation of inputs preserves computability and that all partial functions are computable, ignoring the the input oracle. The latter also implies that total, constant, and everywhere undefined functions are computable.

Lemma 4. *The following functionals mapping relations $R\colon Q\to A\to\mathbb{P}$ to relations of type $I\to O\to\mathbb{P}$ are computable:*

1. $\lambda R\,i\,o.\ F\,R\,(g i)\,o$ *for* $g\colon I\to I'$ *and computable* $F\colon(Q\to A\to\mathbb{P})\to(I'\to O\to\mathbb{P})$,
2. $\lambda R\,i\,o.\ f\,i \triangleright o$ *given* $f\colon I\rightharpoonup O$,
3. $\lambda R\,i\,o.\ f\,i = o$ *given* $f\colon I\to O$,
4. $\lambda R\,i\,o.\ o = v$ *given* $v\colon O$,
5. $\lambda R\,i\,o.\ \bot$.

Proof. For 1, let τ compute F and define $\tau'\,i\,l := \tau\,(g i)\,l$. For 2, define $\tau'\,i\,l := f\,i \ggg\!= \lambda o.\ \mathsf{ret}\ (\mathsf{out}\ o)$, where $\ggg\!=$ is the bind operation of partial functions. All others follow by using (2). □

Next, if $Q = I$ and $A = O$, then the identity functional is computable:

Lemma 5. *The functional mapping $R\colon Q\to A\to\mathbb{P}$ to R itself is computable.*

Proof. Define

$$\tau\,q\,l := \begin{cases} \mathsf{ret}\ (\mathsf{ask}\ q) & \text{if } l = [] \\ \mathsf{ret}\ (\mathsf{out}\ a) & \text{if } l = (q,a) :: l'. \end{cases}$$ □

Moreover, given two functionals and a boolean test on inputs, the process calling either of the two depending on the test outcome is computable:

Lemma 6. *Let F_1 and F_2 both map relations $R\colon Q\to A\to\mathbb{P}$ to relations of type $I\to O\to\mathbb{P}$ and $f\colon I\to\mathbb{B}$. Then F mapping R to the following relation of type $I\to O\to\mathbb{P}$ is computable:*

$$\lambda io. \begin{cases} F_1\ R\ i\ o & \textit{if } f i = \mathsf{true} \\ F_2\ R\ i\ o & \textit{if } f i = \mathsf{false} \end{cases}$$

Proof. Let τ_1 and τ_2 compute F_1 and F_2 respectively and define

$$\tau\, i\, l := \begin{cases} \tau_1\ i\ l & \text{if } f i = \mathsf{true} \\ \tau_2\ i\ l & \text{if } f i = \mathsf{false}. \end{cases}$$ □

Taken together, the previous three lemmas yield computability proofs for functionals consisting of simple operations like calling functions, taking indentities, and branching over conditionals. The next three lemmas extend to partial binding, function composition, and linear search, so in total we obtain an abstraction layer accommodating computability proofs for the most common ingredients of algorithms. As mentioned before, we just state the last three lemmas without proof here and refer to the Coq development and Appendix B for full detail.

Lemma 7. *Let F_1 map relations $R\colon Q{\to}A{\to}\mathbb{P}$ to relations of type $I{\to}O'{\to}\mathbb{P}$, F_2 map relations $R\colon Q{\to}A{\to}\mathbb{P}$ to relations of type $(I \times O'){\to}O{\to}\mathbb{P}$, and both be computable. Then F mapping $R\colon Q{\to}A{\to}\mathbb{P}$ to $\lambda io.\, \exists o'\colon O'.\, F_1\, R\, i\, o' \land F_2\, R\, (i, o')\, o$ of type $I{\to}O{\to}\mathbb{P}$ is computable.*

Lemma 8. *Let F_1 map relations $R\colon Q{\to}A{\to}\mathbb{P}$ to relations $X{\to}Y{\to}\mathbb{P}$, F_2 map relations $R\colon X{\to}Y{\to}\mathbb{P}$ to relations $I{\to}O{\to}\mathbb{P}$, and both be computable. Then F mapping $R\colon Q{\to}A{\to}\mathbb{P}$ to $\lambda io.\ F_2\ (F_1 R)\ i\ o$ of type $I{\to}O{\to}\mathbb{P}$ is computable.*

Lemma 9. *The functional mapping $R\colon (I \times \mathbb{N}){\to}\mathbb{B}{\to}\mathbb{P}$ to the following relation of type $I{\to}\mathbb{N}{\to}\mathbb{P}$ is computable: $\lambda in\ R\ (i, n)\ \mathsf{true} \land \forall m < n.\ R\ (i, m)\ \mathsf{false}$.*

5 Computational Cores of Oracle Computations

In this section, we prove that if F maps $R\colon Q{\to}A{\to}\mathbb{P}$ to a relation $I{\to}O{\to}\mathbb{P}$ and F is computable, then there is a higher-order function $f\colon (Q{\rightharpoonup}A){\to}(I{\rightharpoonup}O)$ such that for any $r\colon Q{\rightharpoonup}A$ with graph R, the graph of $f r$ agrees with $F R$. This means that every computable functional possesses an explicit computational core, mapping (partially) computable input to (partially) computable output, needed for instance to justify that decidability is transported backwards along Turing reductions (Lemma 26).

In preparation, the following two lemmas state simple properties of interrogations regarding concatenation and determinacy. Given $\sigma\colon A^*{\rightharpoonup}Q + O$ and $l\colon A^*$ we write $\sigma@l$ for the sub-tree of σ starting at path l, i.e. for the tree $\lambda l'.\ \sigma(l \mathbin{+\!\!+} l')$.

Lemma 10. *We have interrogations $\sigma\,; R \vdash qs_1\,; as_1$ and $\sigma\ @\ as_1\,; R \vdash qs_2\,; as_2$ if and only if $|qs_2| = |as_2|$ and $\sigma\,; R \vdash qs_1 \mathbin{+\!\!+} qs_2\,; as_1 \mathbin{+\!\!+} as_2$.*

Lemma 11. *Let R be functional and $\sigma\,; R \vdash qs_1\,; as_1$ as well as $\sigma\,; R \vdash qs_2\,; as_2$. Then if $|qs_1| \le |qs_2|$, then qs_1 is a prefix of qs_2 and as_1 is a prefix of as_2.*

Now conveying the main idea, we first define an evaluation function $\delta\ \sigma\ f$: $\mathbb{N} \rightharpoonup Q + O$ which evaluates $\sigma\colon A^*{\rightharpoonup}Q + O$ on $f\colon Q{\rightharpoonup}A$ for at most n questions.

$$\delta\ \sigma\ f\ n := \sigma[\,] \gg\!= \lambda x.\ \begin{cases} \mathsf{ret}\ (\mathsf{out}\ o) & \text{if } x = \mathsf{out}\ o \\ \mathsf{ret}\ (\mathsf{ask}\ q) & \text{if } x = \mathsf{ask}\ q, n = 0 \\ fq \gg\!= \lambda a.\ \delta\ (\sigma\ @\ [a])\ f\ n' & \text{if } x = \mathsf{ask}\ q, n = \mathsf{S}\ n'. \end{cases}$$

The intuition is that δ always reads the initial node of the tree σ by evaluating $\sigma[\,]$. If $\sigma[\,] \triangleright \mathsf{out}\ o$, then δ returns this output. Otherwise, if $\sigma[\,] \triangleright \mathsf{ask}\ q$ and δ has to evaluate no further questions ($n = 0$), it returns $\mathsf{ask}\ q$. If δ has to evaluate $\mathsf{S}\ n$ questions, it evaluates $fq \triangleright a$ and recurses on the subtree of σ with answer a, i.e. on $\sigma \mathbin{@} [a]$. We first verify that δ composes with interrogations by induction on the interrogation:

Lemma 12. *If $\sigma\,; (\lambda qa.\, fq \triangleright a) \vdash qs\,; as$ and $\delta(\tau \mathbin{@} as) fn \triangleright v$ then $\delta\tau fn \triangleright v$.*

Conversely, every evaluation of δ yields a correct interrogation:

Lemma 13. *If $\delta\ \sigma\ f\ n \triangleright \mathsf{out}\ o$ then there are qs and as with $|qs| \leq n$ and $\sigma\,; (\lambda qa.\, fq \triangleright a) \vdash qs\,; as$, and $\sigma\ as \triangleright \mathsf{out}\ o$.*

Proof. By induction on n, using Lemma 10. □

Put together, a computable functional is fully captured by δ for oracles described by partial functions:

Lemma 14. *Given a functional F computed by τ we have that*

$$F(\lambda qa.\, fq \triangleright a)io \leftrightarrow \exists n.\ \delta\ (\tau i)\ f\ n \triangleright \mathsf{out}\ o.$$

This is enough preparation to describe the desired computational core of computable functionals:

Theorem 15. *If F maps $R\colon Q \to A \to \mathbb{P}$ to a relation $I \to O \to \mathbb{P}$ and F is computable, then there is a partial function $f\colon (Q \rightharpoonup A) \to I \rightharpoonup O$ such that if R is computed by a partial function $r\colon Q \rightharpoonup A$, then FR is computed by fr.*

Proof. Let F be computed by τ. We define fri to search for n such that $\delta\ (\tau i)\ f\ n$ returns $\mathsf{out}\ o$, and let it return this o. The claim then follows straightforwardly by the previous lemma and Lemma 11. □

6 Properties of Oracle Semi-Decidability

In the following two sections we establish some standard properties of our synthetic renderings of oracle semi-decidability and Turing reducibility, respectively. All proofs are concise but precise, given that in the synthetic setting they just amount to the essence of the computational manipulations often described just informally for a concrete model of computation in the analytic approach to computability employed e.g. in textbooks.

We first establish the connection to non-relative semi-decidability.

Lemma 16. *If p is semi-decidable, then $\mathcal{S}_q(p)$ for any q.*

Proof. Let $f\colon X \rightharpoonup \mathbb{1}$ be a semi-decider for p. With Lemma 4 (2) the functional mapping R to $\lambda xo.\ fx \triangleright o$ is computable, and it is easily shown to be a semi-decider for p relative to q. □

Lemma 17. *If $\mathcal{S}_q(p)$ and q is decidable, then p is semi-decidable.*

Proof. Let g decide q and let F be a semi-decider of p relative to q. Let f be the function from Theorem 15 that transports computable functions along F. Now $f(\lambda y.\, \mathsf{ret}(gy))$ is a semi-decider for p. □

We next establish closure properties of oracle semi-decidability along reductions. First, we can replace the oracle by any other oracle it reduces to:

Lemma 18. *If $\mathcal{S}_q(p)$ and $q \preceq_\mathsf{T} q'$, then also $\mathcal{S}_{q'}(p)$.*

Proof. Straightforward using Lemma 8. □

Secondly, if we can semi-decide a predicate p relative to q, then also simpler predicates should be semi-decidable relative to q. This however requires a stricter notion of reduction, for instance many-one reductions that rule out complementation. As in [16], we say that $p' : X \to \mathbb{P}$ many-one reduces to $p : Y \to \mathbb{P}$ if there is a function $f : X \to Y$ embedding p' into p:

$$p' \preceq_m p \ := \ \exists f : X \to Y.\, \forall x.\, p'x \leftrightarrow p(fx)$$

Now the sought after property can be stated as follows:

Lemma 19. *If $\mathcal{S}_q(p)$ and $p' \preceq_m p$, then also $\mathcal{S}_q(p')$.*

Proof. Straightforward using Lemma 4 (1,4) and Lemma 7. □

7 Properties of Turing Reducibility

We continue with similarly standard properties of Turing reducibility. Again, all proofs are concise but precise. As a preparation, we first note that Turing reducibility can be characterised without the relational layer.

Lemma 20. *$p \preceq_\mathsf{T} q$ if and only if there is τ such that for all x and b we have*

$$\hat{p}xb \leftrightarrow \exists qs\, as.\ \tau x\, ;\, q \vdash\ qs\, ;\, as \wedge \tau\, x\, as \triangleright \mathsf{out}\ b.$$

Now to begin, we show that Turing reducibility is a preorder.

Theorem 21. *Turing reducibility is reflexive and transitive.*

Proof. Reflexivity follows directly by the identity functional being computable via Lemma 4. Transitivity follows with Lemma 8. □

In fact, Turing reducibility is an upper semilattice:

Theorem 22. *Let $p\colon X \to \mathbb{P}$ and $q\colon Y \to \mathbb{P}$. Then there is a lowest upper bound $p+q\colon X + Y \to \mathbb{P}$ w.r.t. $\preceq_\mathsf{T}$: Let $(p+q)(\mathsf{inl}\ x) := px$ and $(p+q)(\mathsf{inr}\ y) := qy$. then $p+q$ is the join of p and q w.r.t $\preceq_\mathsf{T}$, i.e. $p \preceq_\mathsf{T} p+q$, $q \preceq_\mathsf{T} p+q$, and for all r if $p \preceq_\mathsf{T} r$ and $q \preceq_\mathsf{T} r$ then $p+q \preceq_\mathsf{T} r$.*

Proof. The first two claims follow by Lemma 4 (1) and Lemma 5. For the third, let F_1 reduce p to r and be computed by τ_1 and F_2 reduce q to r computed by τ_2. Define

$$F\,R\,z\,o := \begin{cases} F_1 R\,x\,o & \text{if } z = \mathsf{inl}\ x \\ F_2 R\,x\,o & \text{if } z = \mathsf{inr}\ y \end{cases} \qquad\qquad \tau z l := \begin{cases} \tau_1 x l & \text{if } z = \mathsf{inl}\ x \\ \tau_2 y l & \text{if } z = \mathsf{inr}\ y \end{cases}$$

τ computes F, and F reduces $p + q$ to r. □

We continue by establishing properties analogous to the ones concerning oracle semi-decidability discussed in Section 6. First, analogously to Lemma 16, the non-relativised notion of decidability implies Turing reducibility:

Lemma 23. *If p and $\overline{p}$ are semi-decidable, then $p \preceq_{\mathsf{T}} q$ for any q. In particular, if p is decidable, then $p \preceq_{\mathsf{T}} q$ for any q.*

Proof. Let f semi-decide p and g semi-decide $\overline{p}$. Define $F\,R\,x\,b := \hat{p}xb$ and let τxl ignore l and find the least n such that either $fxn = \mathsf{true}$ or $gxn = \mathsf{true}$ and then return $\mathsf{out}\ (fxn)$. □

Secondly, Lemmas 18 and 19 correspond to the transitivity of Turing reducibility, the latter relying on the fact that many-one reductions induce Turing reductions:

Lemma 24. *If $p \preceq_m q$ then $p \preceq_{\mathsf{T}} q$.*

Proof. Let f be the many-one reduction. Define $F\,R\,x\,b := R\,(fx)\,b$. □

Thirdly, in connection to Lemma 17, we prove the more involved result that Turing reducibility reflects decidability if and only if Markov's principle holds. Markov's principle is an axiom in constructive mathematics stating that satisfiability of functions $\mathbb{N}{\to}\mathbb{B}$ is stable under double negation, i.e.:

$$\mathsf{MP} := \forall f\colon \mathbb{N}{\to}\mathbb{B}.\ \neg\neg(\exists n.\ fn = \mathsf{true}) \to \exists n.\ fn = \mathsf{true}$$

Concretely, MP will be needed as it corresponds to the termination of non-diverging partial functions:

Lemma 25. MP *if and only if $\forall XY. \forall f\colon X \rightharpoonup Y. \forall x.\ \neg\neg(\exists y.\ fx \triangleright Y) \to \exists y.\ fx \triangleright Y$.*

Another ingredient is that total partial function $X \rightharpoonup Y$ induce functions $X \to Y$, as stated here for the specific case of deciders $X \to \mathbb{B}$:

Lemma 26. *Let $f\colon X \rightharpoonup \mathbb{B}$ and $p\colon X {\to} \mathbb{P}$. If $\forall x.\ px \leftrightarrow fx \triangleright \mathsf{true}$ and $\forall x. \exists b.\ fx \triangleright b$, then p is decidable, i.e. there is a function $g\colon X {\to} \mathbb{B}$ such that $\forall x.\ px \leftrightarrow gx = \mathsf{true}$.*

Now assuming $p \preceq_{\mathsf{T}} q$ for q decidable, we can derive a non-diverging partial decider for p, which is turned into a total partial decider with Lemma 25 and then into an actual decider with Lemma 26:

Theorem 27. *Given* MP*, if q is decidable and $p \preceq_\mathsf{T} q$, then p is decidable.*

Proof. Let F be the reduction relation and let f transport computability along it as in Theorem 15. Let g decide q. It is straightforward that $\forall xb.\ \hat{p}xb \leftrightarrow f(\lambda y.\mathsf{ret}\ (gy))x \triangleright b$ (*). It suffices to prove that $\forall x.\exists b.\ f(\lambda y.\mathsf{ret}\ (gy))x \triangleright b$ to obtain the claim from Lemma 26.

Using Lemma 25 and MP, given x it suffices to prove $\neg\neg\exists b.\ f(\lambda y.\mathsf{ret}\ (gy))x \triangleright b$. Because the goal is negative and we can prove $\neg\neg(px \lor \neg px)$, we are allowed to do a case analysis on px. In both cases we can prove termination using (*). □

As hinted above, the previous theorem could be stated without MP by using a notion of decidability via a non-diverging partial decider $f : X \rightharpoonup \mathbb{B}$, i.e. with $\forall x.\neg\neg\exists b.\ fx \triangleright b$. However, in the stated form, it is in fact equivalent to MP:

Lemma 28. *If p is decidable if there is decidable q with $p \preceq_\mathsf{T} q$, then* MP *holds.*

Proof. By [16, Theorem 2.20] it suffices to prove that whenever $p : \mathbb{N} \to \mathbb{P}$ and $\overline{p}$ are semi-decidable, then also p is decidable, which follows by Lemma 23 and the assumption for some choice of a decidable predicate q. □

Lastly, we prove that using classical logic, predicates are Turing-equivalent to their complement, providing evidence for the inherent classicality:

Lemma 29. *For double-negation stable p, $p \preceq_\mathsf{T} \overline{p}$ and $\overline{p} \preceq_\mathsf{T} p$.*

Proof. Assume $\forall x.\ \neg\neg px \to px$. For both reductions, take $F R\, x\, b := R\, x\, (\neg_\mathbb{B} b)$, which is computable by Lemma 7, Lemma 5, and Lemma 4 (1,3). □

Lemma 30. *Let X be some type with $x_0 : X$. If $p \preceq_\mathsf{T} \overline{p}$ for all $p : X \to \mathbb{P}$, then* MP *implies the law of excluded middle* ($\mathsf{LEM} := \forall P : \mathbb{P}.\ P \lor \neg P$).

Proof. Assume MP, X with $x_0 : X$, and that $p \preceq_\mathsf{T} \overline{p}$ for all $p : X \to \mathbb{P}$. It suffices to prove that for every proposition P we have $\neg\neg P \to P$. So assume $\neg\neg P$.

By MP and Theorem 27, we have that whenever $\lambda x.\ \neg P$ is decidable, then so is $\lambda x.\ P$. Now since $\neg\neg P$ holds, $\lambda x.\ \mathsf{false}$ decides $\lambda x.\ \neg P$. Thus we have a decider f for $\lambda x.\ P$. A case analysis on $f x_0$ yields either P and we are done – or $\neg P$, which is ruled out by $\neg\neg P$. □

The last lemma ensures that some amount of classical logic is necessary to prove that Turing reducibility is closed under complements, since it is well-known that MP does not imply LEM.

8 Turing Reducibility and Truth-Table Reducibility

As a further expectable property, we establish the well-known connection of Turing reducibility to truth-table reducibility [35], namely that every truth-table reduction induces a Turing reduction while the converse does not hold. Note that the proofs in this section have a classical flavour where explicitly mentioned.

Intuitively, a truth-table reduction can be seen as a restricted form of a Turing reduction: to reduce a predicate $p\colon X\to\mathbb{P}$ to a predicate $q\colon Y\to\mathbb{P}$, on input x, it has to compute a list of oracle queries of type Y^* and provide a truth-table mapping the list of answers of the oracle for q to an output of the reduction. Consequently, questions can not depend on answers of the oracle, and no non-termination is permitted. See also the explanations by Rogers [38, §8.3] or Odifreddi [31, III.3].

Concretely, we use the synthetic definition of truth-table reducibility from Forster and Jahn [13]. We model truth-tables as lists $\mathbb{B}^*$, but just work with a boolean evaluation predicate $l \vDash T$ and refer to the Coq code for its definition.

$$p \preceq_{\mathtt{tt}} q := \exists f\colon X\to Y^*\times\mathbb{B}^*. \forall x\colon X. \forall l\colon \mathbb{B}^*. \, \mathsf{Forall}_2\, \hat{q}\, (\pi_1(fx))\, l \to (px \leftrightarrow l \vDash \pi_2(fx))$$

where Forall_2 lifts binary predicates to lists pointwise by conjunction.

We first show that truth-table reducibility implies Turing reducibility.

Theorem 31. *If q is classical (i.e. $\forall y.\ qy \lor \neg qy$), then $p \preceq_{\mathtt{tt}} q$ implies $p \preceq_{\mathsf{T}} q$.*

Proof. Let f be the truth-table reduction. Define F to map $R\colon Y\to\mathbb{B}\to\mathbb{P}$ to

$$\lambda xb.\ \exists l\colon \mathbb{B}^*.\ \mathsf{Forall}_2\ R\ (\pi_1(fx))\ l \land l \vDash \pi_2(fx))$$

which can be computed by the tree

$$\tau x l := \begin{cases} \mathsf{ret}\ (\mathsf{ask}\ a) & \text{if } \pi_1(fx) \text{ at position } |l| \text{ is } a \\ \mathsf{ret}\ (\mathsf{out}\ (l \vDash \pi_2(fx))) & \text{otherwise.} \end{cases}$$

The direction from right to left is straightforward. For the direction from left to right, it suffices to prove the existence of l with $\mathsf{Forall}_2\ \hat{q}\ \pi_1(fx)\ l$, following by induction on $\pi_1(fx)$, using the assumption that q is classical to construct l. □

We now prove that the inclusion of truth-table reducibility in Turing reducibility is strict. Forster and Jahn [13] introduce a hypersimple predicate $H_I\colon\mathbb{N}\to\mathbb{P}$ as the deficiency predicate of a strongly enumerable predicate $I\colon\mathbb{N}\to\mathbb{P}$ [8]: Given an injective, strong enumerator E_I of I ($\forall x.\ Ix \leftrightarrow \exists n. E_I n = x$), they set

$$H_I x \ := \ \exists x_0 > x.\ E_I x_0 < E_I x.$$

They prove that I does not truth-table reduce to H_I assuming axioms for synthetic computability, and in particular that the halting problem fulfills the preconditions for I. Thus, to separate truth-table from Turing reducibility, it suffices to give a Turing reduction $I \preceq_{\mathsf{T}} H_I$ (without having to assume axioms for synthetic computability).

Algorithmically, one can decide Iz given a partial function $f\colon\mathbb{N}\rightharpoonup\mathbb{B}$ deciding H_I as follows: We search for x such that $fx \triangleright \mathsf{false}$ and $E_I x > z$, i.e. $\neg H_I x$. Such an x does (not not) exists because the complement of H_I is non-finite. Then Iz holds if and only if $z \in [E_I 0, \dots E_I(x+1)]$.

Formally, we first establish the classical existence of such x in the more general situation of arbitrary non-finite predicates and injections.

Lemma 32. *If* $p : X \to \mathbb{P}$ *is non-finite and* $f : X \to \mathbb{N}$ *is injective, then for* $z : \mathbb{N}$

$$\neg\neg\exists x.\, px \land fx \geq z \land \forall y.\, py \to fy \geq z \to fx \leq fy.$$

Next, we verify the resulting characterisation of I via list membership.

Lemma 33. *If* $\neg H_I x$ *and* $E_I x > z$*, then* $Iz \leftrightarrow [E_I 0, \ldots, E_I(x+1)]$.

Put together, we can describe the desired Turing reduction.

Theorem 34. *Assuming* LEM*, if* I *is strongly enumerable, then* $I \preceq_{\mathsf{T}} H_I$.

Proof. We define F to map relations R to the relation

$$\lambda zb.\, \exists x.\, R\, x\, \mathsf{false} \;\land\; E_I x > z \;\land\; (b = \mathsf{true} \leftrightarrow z \in [E_I 0, \ldots, E_I(x+1)]) \\ \land\; (\forall x' < x.\, (R\, x'\, \mathsf{true} \lor (R\, x'\, \mathsf{false} \land E_I x' \leq z)))$$

which is straightforward to show computable.

Regarding $F(\hat{H}_I)zb \leftrightarrow \hat{I}zb$, the direction from left to right is immediate from Lemma 33. For the direction from right to left, assume $\hat{I}zb$. Let x be obtained for H_I and E_I from Lemma 32. Then x fulfils the claim by Lemma 33. □

Since in this paper we do not assume axioms for synthetic computability that imply $I \not\preceq_{\mathtt{tt}} H_I$, we keep the conclusion that truth-table reducibility is strictly stronger than Turing reducibility implicit.

9 Post's Theorem (PT)

There are various results (rightly) called "Post's theorem" in the literature. Here, we are concerned with the result that if both a predicate and its complement are semi-decidable, the predicate is decidable. This theorem was proved by Post in 1944 [35], and is not to be confused with Post's theorem relating the arithmetical hierarchy and Turing jumps from 1948 [36]. We thus simply refer to the result we consider as PT_0, and use PT for its relativised version.

It is well-known that PT_0 is equivalent to Markov's principle [42,1,16]. We here prove that the relativised version PT is fully constructive, and that in fact the equivalence proof of MP and PT_0 can be given using PT and the already proven equivalence between MP and the statement that Turing reducibility transports decidability backwards given in Section 7.

As an auxiliary notion, we introduce an equivalent but a priori more expressive form of interrogations which maintains an internal state of the computation and can "stall", i.e. trees do not have to either ask a question or produce an output, but can alternatively choose to just update the state. Such trees are of type $S \to A^* \rightharpoonup (S \times Q^?) + O$, where $Q^?$ is the inductive option type with elements None and $\mathsf{Some}\; q$ for $q : Q$.

A stalling tree is a partial function $\sigma\colon S \to A^* \rightharpoonup (S \times Q^?) + O$. We define a stalling interrogation predicate $\sigma \,;\, R \vdash qs \,;\, as \,;\, s \succ s'$ inductively by:

$$\frac{}{\sigma \,;\, R \vdash [\,] \,;\, [\,] \,;\, s \succ s} \qquad \frac{\sigma \,;\, R \vdash qs \,;\, as \,;\, s \succ s'' \qquad \sigma \;\, ;\, s'' \;\, ;\, as \triangleright \mathsf{ask}\ (s', \mathsf{None})}{\sigma \,;\, R \vdash qs \,;\, as \,;\, s \succ s'}$$

$$\frac{\sigma \,;\, R \vdash qs \,;\, as \,;\, s \succ s'' \qquad \sigma \;\, ;\, s'' \;\, ;\, as \triangleright \mathsf{ask}\ (s', \mathsf{Some}\ q) \qquad Rqa}{\sigma \,;\, R \vdash qs +\!\!+ [q] \,;\, as +\!\!+ [a] \,;\, s \succ s'}$$

The first and third rule are not significantly different from before, apart from also threading a state s. The second rule allows the tree to stall by only updating the state to s', but without asking an actual question. Intuitively, we can turn a stalling tree τ into a non-stalling one τ' by having τ' compute on input as first all results of τ on all prefixes of as, starting from a call $\tau\, i\, s_0\ as$ for a given initial state s_0. We give this construction in full detail in Appendix B.

A functional F mapping $R\colon Q \to A \to \mathbb{P}$ to a relation of type $I \to O \to \mathbb{P}$ is computable via stalling interrogations if there are a type S, an element $s_0\colon S$, and a function $\tau\colon I \to S \to A^* \rightharpoonup (S \times Q^?) + O$ such that

$$\forall R\, i\, o.\ F R\, i\, o \leftrightarrow \exists qs\ as\ s.\ \tau i \,;\, R \vdash qs \,;\, as \,;\, s_0 \succ s \ \wedge\ \tau\, i\, s\ as \triangleright \mathsf{out}\ o.$$

We prove that the two definitions of computability are equivalent in Appendix B and immediately move on to the proof of PT.

Theorem 35. (PT) *If* $\mathcal{S}_q(p)$ *and* $\mathcal{S}_q(\overline{p})$, *then* $p \preceq_{\mathsf{T}} q$.

Proof. Let $p\colon X \to \mathbb{P}$ and $q\colon Y \to \mathbb{P}$ as well as F_1 and F_2 be the functionals representing the semi-deciders, computed respectively by τ_1 and τ_2. The intuition is, on input x and as, to execute $\tau_1\ x$ and $\tau_2\ x$ in parallel and ensure that both their questions are asked. The interrogation can finish with true if $\tau_1\ x$ outputs a value, and with false if $\tau_2\ x$ does.

There are two challenges in making this intuition formal as an oracle computation: Only answers from as that τ_1 and τ_2 asked for have to be actually passed to it, respectively, and both τ_1 and τ_2 need to be allowed to ask all of their questions and eventually produce an output fairly, even though only one of them ever will.

Using Lemma 20, we define the Turing reduction without providing the relational layer and instead directly construct a tree τ based on stalling interrogations with state type $S := Y^? \times \mathbb{N} \times (\mathbb{B} \times Y)^*$. The first argument is used to remember a question that needs to be asked next, arising from cases where both τ_1 and τ_2 want to ask a question. The second argument is a step-index n used to evaluate both τ_1 and τ_2 for n steps. The third argument records which question was asked by τ_1 and which by τ_2. To then construct τ compactly, we define helper functions $\mathsf{getas}_{1,2}\colon (\mathbb{B} \times Y)^* \to \mathbb{B}^* \to Y^*$ which choose answers from the second list according to the respective boolean in the first list.

We then define

$$\tau(\mathsf{Some}\ q, n, t)\, as := \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{None}, n, t +\!\!+ [(\mathsf{false}, q)], \mathsf{Some}\ q))$$

$$\tau(\mathsf{None}, n, t)\, as := \begin{cases} \mathsf{ret}\ (\mathsf{out\ true}) & \text{if } x_1 = \mathsf{Some}\ (\mathsf{out}\ o) \\ \mathsf{ret}\ (\mathsf{out\ false}) & \text{if } x_2 = \mathsf{Some}\ (\mathsf{out}\ o) \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{Some}\ q', \mathsf{S}\ n, t +\!\!+ [(\mathsf{true}, q)], \mathsf{Some}\ q)) & \text{if } x_1 = \mathsf{Some}\ (\mathsf{ask}\ q) \\ & \text{ and } x_2 = \mathsf{Some}\ (\mathsf{ask}\ q') \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{None}, \mathsf{S}\ n, t +\!\!+ [(\mathsf{true}, q)], \mathsf{Some}\ q)) & \text{if } x_1 = \mathsf{Some}\ (\mathsf{ask}\ q) \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{None}, \mathsf{S}\ n, t +\!\!+ [(\mathsf{false}, q)], \mathsf{Some}\ q)) & \text{if } x_2 = \mathsf{Some}\ (\mathsf{ask}\ q) \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{None}, \mathsf{S}\ n, t, \mathsf{None})) & \text{otherwise} \end{cases}$$

where $x_1 = \rho^n\ (\tau_1\, x\, (\mathsf{getas}_1\ t\ as))$ and $x_2 = \rho^n\ (\tau_2\, x\, (\mathsf{getas}_2\ t\ as))$, with ρ being a step-indexed evaluation function for partial values.

This means that whenever τ_1 returns an output, then true is returned and whenever τ_2 returns an output, then false is returned while no question is ever missed and the interrogation stalls if n does not suffice to evaluate either τ_1 or τ_2. The invariants to prove that this indeed yields the wanted Turing reduction are technical but pose no major hurdles, we refer to the Coq code for details. □

Corollary 36. *The following are equivalent:*

1. MP
2. *Termination of partial functions is double negation stable.*
3. *Turing reducibility transports decidability backwards.*
4. PT_0

Proof. Implications (1) → (2) and (4) → (1) are well-known. We have already proved implication (2) → (3). It suffices to prove (3) → (4), which is almost direct using PT: Assume that for all X, Y, $p\colon X \to \mathbb{P}$, and $q\colon Y \to \mathbb{P}$ we have that if q is decidable and $p \preceq_{\mathsf{T}} q$, then p is decidable. Let furthermore p and its complement be semi-decidable. We prove that p is decidable. Clearly, it suffices to prove that $p \preceq_{\mathsf{T}} q$ for a decidable predicate q (e.g. $\lambda n\colon \mathbb{N}. \top$). Using PT, it suffices to prove p and its complement semi-decidable in q, which in turn follows from the assumption that they are semi-decidable and Lemma 16. □

10 Discussion

Mechanisation in Coq The Coq mechanisation accompanying this paper closely follows the structure of the hyperlinked mathematical presentation and spans roughly 2500 lines of code for the novel results, building on a library of basic synthetic computability theory. It showcases the feasibility of mechanising ongoing research with reasonable effort and illustrates the interpretation of synthetic oracle computations as a natural notion available in dependently-typed programming languages. In fact, using Coq helped us a lot with finding the proofs concerning constructive reverse mathematics (Lemmas 28 and 30 and Corollary 36) in the first place, where subtleties like double negations need to be tracked over small changes in the definitions.

On top of the usual proof engineering, we used three notable mechanisation techniques. First, we generalise over all possible implementations of partial functions, so our code is guaranteed to just rely on the abstract interface described in

Appendix A. Secondly, we devised a custom tactic `psimpl` that simplifies goals involving partial functions by strategically rewriting with the specifications of the respective operations. Thirdly, to establish computability of composed functionals, instead of constructing a complicated tree at once, we postpone the construction with the use of existential variables and apply abstract lemmas such as the ones described in Section 4 to obtain the trees step by step.

Related Work Synthetic computability was introduced by Richman [37] and popularised by Richman, Bridges, and Bauer [5,1,2,3]. In synthetic computability, one assumes axioms such as CT ("Church's thesis" [28,42]), postulating that *all* functions are μ-recursive. CT is proved consistent for univalent type theory by Swan and Uemura [39]. Since univalent type theory proves unique choice, using it as the basis for computability theory renders CT inconsistent with already the weak principle of omniscience [10], and consequently with the law of excluded middle, precluding interesting results in constructive reverse mathematics.

Forster [12] identifies that working in CIC allows assuming CT and its consequences even under the presence of the law of excluded middle. This approach has been used to develop the theory of many-one and truth-table reducibility [13], to give a proof of the Myhill isomorphism theorem [14] and a more general treatment of computational back-and-forth arguments [21], to show that random numbers defined using Kolmogorov complexity form a simple set [17], to analyse Tennenbaum's theorem regarding its constructive content [20], to give computational proofs of Gödel's first incompleteness theorem [23,24], and to develop an extensive Coq library of undecidability proofs [18].

The first synthetic definition of oracle computability is due to Bauer [3], based on continuous functionals in the effective topos. The first author introduced a classically equivalent definition in his PhD thesis [11] based on joint work with the second author. Subsequently, we have adapted this definition into one constructively equivalent to Bauer's definition [15]. All these previous definitions however have in common that it is unclear how to derive an enumeration of all oracle computable functionals from CT as used in [22,30], because they do not reduce higher-order functionals to first-order functions. Recently, Swan has suggested a definition of oracle computability based on modalities in univalent type theory [40].

Future Work With the present paper, we lay the foundation for several future investigations concerning synthetic oracle computability in the context of axioms like CT, both by improving on related projects and by tackling new challenges. First, a rather simple test would be the Kleene-Post theorem [27], establishing incomparable Turing degrees as already approximated in [22], assuming an enumeration of all oracle computations of their setting. Similarly, we plan to establish Post's theorem [36], connecting the arithmetical hierarchy with Turing degrees. An interesting challenge would be a synthetic proof of the Friedberg-Muchnik theorem [19,29], solving Post's problem [35] concerning the existence of undecidable Turing degrees strictly below the halting problem.

Acknowledgements We want to thank Felix Jahn, Gert Smolka, Dominique Larchey-Wendling, and the participants of the TYPES '22 conference for many fruitful discussions about Turing reducibility, as well as Martin Baillon, Yann Leray, Assia Mahboubi, Pierre-Marie Pédrot, and Matthieu Piquerez for discussions about notions of continuity. The central inspiration to start working on Turing reducibility in type theory is due to Andrej Bauer's talk at the Wisconsin logic seminar in February 2021.

References

1. Bauer, A.: First steps in synthetic computability theory. Electronic Notes in Theoretical Computer Science **155**, 5–31 (2006). `https://doi.org/10.1016/j.entcs.2005.11.049`
2. Bauer, A.: On fixed-point theorems in synthetic computability. Tbilisi Mathematical Journal **10**(3), 167–181 (2017). `https://doi.org/10.1515/tmj-2017-0107`
3. Bauer, A.: Synthetic mathematics with an excursion into computability theory (slide set). University of Wisconsin Logic seminar (2020), `http://math.andrej.com/asset/data/madison-synthetic-computability-talk.pdf`
4. Bauer, A.: Formalizing invisible mathematics. In: Workshop on Machine Assisted Proofs, Institute for Pure and Applied Mathematics (IPAM) at the University of California in Los Angeles (UCLA), February 13 – 17, 2023 (2023), `https://www.youtube.com/watch?v=wZSvuCJBaFU`
5. Bridges, D., Richman, F.: Varieties of constructive mathematics, vol. 97. Cambridge University Press (1987). `https://doi.org/10.1017/CBO9780511565663`
6. Coquand, T., Huet, G.P.: The calculus of constructions. Information and Computation **76**(2/3), 95–120 (1988). `https://doi.org/10.1016/0890-5401(88)90005-3`, `https://doi.org/10.1016/0890-5401(88)90005-3`
7. Davis, M.D.: Computability and Unsolvability. McGraw-Hill Series in Information Processing and Computers, McGraw-Hill (1958)
8. Dekker, J.C.E.: A theorem on hypersimple sets. Proceedings of the American Mathematical Society **5**, 791–796 (1954). `https://doi.org/10.1090/S0002-9939-1954-0063995-6`
9. Escardo, M.: Continuity of Gödel's system T definable functionals via effectful forcing. Electronic Notes in Theoretical Computer Science **298**, 119–141 (11 2013). `https://doi.org/10.1016/j.entcs.2013.09.010`
10. Forster, Y.: Church's Thesis and Related Axioms in Coq's Type Theory. In: Baier, C., Goubault-Larrecq, J. (eds.) 29th EACSL Annual Conference on Computer Science Logic (CSL 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 183, pp. 21:1–21:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). `https://doi.org/10.4230/LIPIcs.CSL.2021.21`, `https://drops.dagstuhl.de/opus/volltexte/2021/13455`
11. Forster, Y.: Computability in Constructive Type Theory. Ph.D. thesis, Saarland University (2021). `https://doi.org/10.22028/D291-35758`
12. Forster, Y.: Parametric Church's Thesis: Synthetic computability without choice. In: International Symposium on Logical Foundations of Computer Science. pp. 70–89. Springer (2022). `https://doi.org/10.1007/978-3-030-93100-1_6`
13. Forster, Y., Jahn, F.: Constructive and Synthetic Reducibility Degrees: Post's Problem for Many-One and Truth-Table Reducibility in Coq. In: Klin, B., Pimentel, E. (eds.) 31st EACSL Annual Conference on Computer Science Logic

(CSL 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 252, pp. 21:1–21:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.CSL.2023.21, https://drops.dagstuhl.de/opus/volltexte/2023/17482

14. Forster, Y., Jahn, F., Smolka, G.: A Computational Cantor-Bernstein and Myhill's Isomorphism Theorem in Constructive Type Theory. In: CPP 2023 - 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 1–8. ACM, Boston, United States (Jan 2023). https://doi.org/10.1145/3573105.3575690, https://inria.hal.science/hal-03891390
15. Forster, Y., Kirst, D.: Synthetic Turing reducibility in constructive type theory. 28th International Conference on Types for Proofs and Programs (TYPES 2022) (2022), https://types22.inria.fr/files/2022/06/TYPES_2022_paper_64.pdf
16. Forster, Y., Kirst, D., Smolka, G.: On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019. ACM Press (2019). https://doi.org/10.1145/3293880.3294091, https://doi.org/10.1145/3293880.3294091
17. Forster, Y., Kunze, F., Lauermann, N.: Synthetic Kolmogorov Complexity in Coq. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving (ITP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 237, pp. 12:1–12:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.12, https://drops.dagstuhl.de/opus/volltexte/2022/16721
18. Forster, Y., Larchey-Wendling, D., Dudenhefner, A., Heiter, E., Kirst, D., Kunze, F., Smolka, G., Spies, S., Wehr, D., Wuttke, M.: A Coq library of undecidable problems. In: The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020). (2020), https://github.com/uds-psl/coq-library-undecidability
19. Friedberg, R.M.: Two recursively enumerable sets of incomparable degrees of unsovlability (solution of Post's problem), 1944. Proceedings of the National Academy of Sciences **43**(2), 236–238 (Feb 1957). https://doi.org/10.1073/pnas.43.2.236, https://doi.org/10.1073/pnas.43.2.236
20. Hermes, M., Kirst, D.: An analysis of Tennenbaum's theorem in constructive type theory. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 228, pp. 9:1–9:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.9, https://doi.org/10.4230/LIPIcs.FSCD.2022.9
21. Kirst, D.: Computational Back-And-Forth Arguments in Constructive Type Theory. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving (ITP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 237, pp. 22:1–22:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.22, https://drops.dagstuhl.de/opus/volltexte/2022/16731
22. Kirst, D., Forster, Y., Mück, N.: Synthetic Versions of the Kleene-Post and Post's Theorem. 28th International Conference on Types for Proofs and Programs (TYPES 2022) (2022), https://types22.inria.fr/files/2022/06/TYPES_2022_paper_65.pdf

23. Kirst, D., Hermes, M.: Synthetic undecidability and incompleteness of first-order axiom systems in Coq: Extended version. Journal of Automated Reasoning **67**(1), 13 (2023). https://doi.org/10.1007/s10817-022-09647-x
24. Kirst, D., Peters, B.: Gödel's theorem without tears - essential incompleteness in synthetic computability. In: Klin, B., Pimentel, E. (eds.) 31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland. LIPIcs, vol. 252, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.CSL.2023.30, https://doi.org/10.4230/LIPIcs.CSL.2023.30
25. Kleene, S.C.: Recursive functionals and quantifiers of finite types i. Transactions of the American Mathematical Society **91**(1), 1 (Apr 1959). https://doi.org/10.2307/1993145, https://www.jstor.org/stable/1993145?origin=crossref
26. Kleene, S.C.: Introduction to metamathematics, vol. 483. van Nostrand New York (1952)
27. Kleene, S.C., Post, E.L.: The upper semi-lattice of degrees of recursive unsolvability. The Annals of Mathematics **59**(3), 379 (May 1954). https://doi.org/10.2307/1969708, https://doi.org/10.2307/1969708
28. Kreisel, G.: Mathematical logic. Lectures in modern mathematics **3**, 95–195 (1965). https://doi.org/10.2307/2315573
29. Muchnik, A.A.: On strong and weak reducibility of algorithmic problems. Sibirskii Matematicheskii Zhurnal **4**(6), 1328–1341 (1963)
30. Mück, N.: The Arithmetical Hierarchy, Oracle Computability, and Post's Theorem in Synthetic Computability. Bachelor's thesis, Saarland University (2022), https://ps.uni-saarland.de/~mueck/bachelor.php
31. Odifreddi, P.: Classical recursion theory: The theory of functions and sets of natural numbers. Elsevier (1992)
32. van Oosten, J.: A combinatory algebra for sequential functionals of finite type. In: Models and Computability, pp. 389–406. Cambridge University Press (jun 1999). https://doi.org/10.1017/cbo9780511565670.019
33. van Oosten, J.: Partial combinatory algebras of functions. Notre Dame Journal of Formal Logic **52**(4), 431–448 (2011). https://doi.org/10.1215/00294527-1499381
34. Paulin-Mohring, C.: Inductive definitions in the system Coq rules and properties. In: International Conference on Typed Lambda Calculi and Applications. pp. 328–345. Springer (1993). https://doi.org/10.1007/BFb0037116
35. Post, E.L.: Recursively enumerable sets of positive integers and their decision problems. bulletin of the American Mathematical Society **50**(5), 284–316 (1944). https://doi.org/10.1090/S0002-9904-1944-08111-1
36. Post, E.L.: Degrees of recursive unsolvability - preliminary report. In: Bulletin of the American Mathematical Society. vol. 54:7, pp. 641–642. American Mathematical Society (AMS) (1948)
37. Richman, F.: Church's thesis without tears. The Journal of symbolic logic **48**(3), 797–803 (1983). https://doi.org/10.2307/2273473
38. Rogers, H.: Theory of recursive functions and effective computability (1987)
39. Swan, A., Uemura, T.: On Church's thesis in cubical assemblies. arXiv preprint arXiv:1905.03014 (2019), https://arxiv.org/abs/1905.03014
40. Swan, A.W.: Oracle modalities. Second International Conference on Homotopy Type Theory (HoTT 2023) (2023), https://hott.github.io/HoTT-2023/abstracts/HoTT-2023_abstract_35.pdf
41. The Coq Development Team: The Coq proof assistant (Jun 2023). https://doi.org/10.5281/zenodo.8161141, https://doi.org/10.5281/zenodo.8161141

42. Troelstra, A.S., van Dalen, D.: Constructivism in mathematics. vol. i. Studies in Logic and the Foundations of Mathematics **26** (1988)
43. Turing, A.M.: Systems of logic based on ordinals. Proceedings of the London mathematical society **2**(1), 161–228 (1939). https://doi.org/10.1112/plms/s2-45.1.161

A Glossary of Definitions

We collect some basic notations and definitions:

- $\mathbb{P}$ is the (impredicative) universe of propositions.
- *Natural numbers:* $n : \mathbb{N} ::= 0 \mid \mathsf{S}\, n$
- *Booleans:* $b : \mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$
- *Unit type:* $\mathbb{1} ::= \star$
- *Sum type:* $X + Y ::= \mathsf{inl} x \mid \mathsf{inr} y \quad (x : X, y : Y)$
- *Option type:* $o : X^? ::= \mathsf{None} \mid \mathsf{Some}\, x \quad (x : X)$
- *Lists:* $l : X^* ::= [\,] \mid x :: l \quad (x : X)$

List operations We often rely on concatenation of of two lists $l_1 +\!\!+ l_2$:

$$[\,] +\!\!+ l_2 := l_2 \qquad\qquad (x :: l_1) +\!\!+ l_2 := x :: (l_1 +\!\!+ l_2)$$

Also, we use an inductive predicate $\mathsf{Forall}_2 \colon (X \to Y \to \mathbb{P}) \to X^* \to Y^* \to \mathbb{P}$

$$\frac{}{\mathsf{Forall}_2\, p\, [\,]\, [\,]} \qquad\qquad \frac{p x y \qquad \mathsf{Forall}_2\, p\, l_1\, l_2}{\mathsf{Forall}_2\, p\, (x :: l_1)\, (y :: l_2)}$$

Characteristic relation The characteristic relation $\hat{p} \colon X \to \mathbb{B} \to \mathbb{P}$ of a predicate $p \colon X \to \mathbb{P}$ is introduced in Section 3 as

$$\hat{p} \;:=\; \lambda x b. \begin{cases} px & \text{if } b = \mathsf{true} \\ \neg px & \text{if } b = \mathsf{false}. \end{cases}$$

Reducibility $\preceq_m$ is many-one reducibility, introduced in Section 6. $\preceq_{\mathtt{tt}}$ is truth-table reducibility, introduced in Section 8. $\preceq_{\mathsf{T}}$ is Turing reducibility, introduced in Section 3.

Interrogations The interrogation predicate $\sigma\, ; R \vdash\, qs\, ;\, as$ is introduced in Section 2. It works on a tree $\sigma \colon A^* \to Q + O$. We often also use trees taking an input, i.e. $\tau \colon I \to A^* \to Q + O$. Given σ, we denote the subtree starting at path $l \colon A^*$ with $\sigma \,@\, l := \lambda l'.\, \sigma(l +\!\!+ l')$.

Partial functions We use an abstract type of partial values over X, denoted as $\mathcal{P}X$, with evaluation relation $\triangleright : \mathcal{P}X \to X \to \mathbb{P}$. We set $X \rightharpoonup Y := X \to \mathcal{P}Y$ and use

- $\mathsf{ret} : X \rightharpoonup X$ with $\mathsf{ret}\ x \triangleright x$,
- $\gg\!= : \mathcal{P}X \to (X \to \mathcal{P}Y) \to \mathcal{P}Y$ with $x \gg\!= f \triangleright y \leftrightarrow \exists v.\ x \triangleright v \wedge f v \triangleright y$,
- $\mu : (\mathbb{N} \to \mathcal{P}\mathbb{B}) \to \mathcal{P}\mathbb{N}$ with $\mu f \triangleright n \leftrightarrow f n \triangleright \mathsf{true} \wedge \forall m < n.\ f m \triangleright \mathsf{false}$, and
- $\mathsf{undef} : \mathcal{P}X$ with $\forall v.\ \mathsf{undef} \not\triangleright v$.

One can for instance implement $\mathcal{P}X$ as monotonic sequences $f : \mathbb{N} \to X^?$, i.e. with $f n = \mathsf{Some}\ x \to \forall m \geq n.\ f m = \mathsf{Some}\ x$ and $f \triangleright x := \exists n.\ f n = \mathsf{Some}\ x$. For any implementation it is only crucial that the graph relation $\lambda xy. f x \triangleright y$ for $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is semi-decidable but cannot be proved decidable. Semi-decidability induces a function $\rho : \mathcal{P}X \to \mathbb{N} \to X^?$, which we write as $\rho^n x$ with the properties that $x \triangleright v \leftrightarrow \exists n.\ \rho^n x = \mathsf{Some}\ v$ and $\rho^n x = \mathsf{Some}\ v \to \forall m \geq n.\ \rho^m x = \mathsf{Some}\ v$.

B Extended Forms of Interrogations

B.1 Extended Interrogations with State

As an auxiliary notion, before introducing the stalling interrogations, we first introduce extended interrogations with a state argument, but without stalling. An extended tree is a function $\sigma : S \to A^* \rightharpoonup (S \times Q) + O$. We define an inductive extended interrogation predicate $\sigma\ ; R \vdash qs\ ; as\ ; s \succ s'$ by:

$$\frac{}{\sigma\ ; R \vdash [\,]\ ; [\,]\ ; s \succ s} \qquad \frac{\sigma\ ; R \vdash qs\ ; as\ ; s \succ s'' \qquad \sigma\ s''\ as \triangleright \mathsf{ask}\ (s', q) \qquad R q a}{\sigma\ ; R \vdash qs \mathbin{+\!\!+} [q]\ ; as \mathbin{+\!\!+} [a]\ ; s \succ s'}$$

A functional F mapping $R : Q \to A \to \mathbb{P}$ to a relation of type $I \to O \to \mathbb{P}$ is computable via extended interrogations if there are a type S, an element $s_0 : S$, and a function $\tau : I \to S \to A^* \rightharpoonup (S \times Q) + O$ such that

$$\forall R\, i\, o.\ F R\, i\, o \leftrightarrow \exists qs\ as\ s.\ \tau i\ ; R \vdash qs\ ; as\ ; s_0 \succ s\ \wedge\ \tau\, i\, s\ as \triangleright \mathsf{out}\ o.$$

Note that we do not pass the question history to the function here, because if necessary it can be part of the type S.

Lemma 37. *Computable functionals are computable via extended interrogations.*

Proof. Let F be computable by τ. Set S to be any inhabited type with element s_0 and define

$$\tau'\ i\ s\ l := \tau\, i\, l \gg\!= \lambda x. \begin{cases} \mathsf{ret}\ (\mathsf{ask}\ (s, q)) & \text{if } x = \mathsf{ask}\ q \\ \mathsf{ret}\ o & \text{if } x = \mathsf{out}\ o. \end{cases}.$$

Then τ' computes F via extended interrogations. ☐

Lemma 38. *Functionals computable via extended interrogations are computable.*

Proof. Let $\tau \colon I \to S \to A^* \rightharpoonup (S \times Q) + O$ compute F via extended interrogations. Define $\tau' \colon S \to A^* \to I \to A^* \rightharpoonup Q + O$ as

$$\tau' \; s \; l \; i \; [] := \tau \; i \; s \; l \gg\!= \begin{cases} \mathsf{ret}\ (\mathsf{ask}\ q) & \text{if } x = \mathsf{ask}\ (e, q) \\ \mathsf{ret}\ (\mathsf{out}\ o) & \text{if } x = \mathsf{out}\ o, \end{cases}$$

$$\tau' \, s \, l \, i \, (a :: as) := \tau \, s \, l \, i \gg\!= \lambda x. \begin{cases} \tau' \, s' \, (l \mathbin{+\!\!+} [a]) \, i \, as & \text{if } x = \mathsf{ask}\ (s', q) \\ \mathsf{ret}\ (\mathsf{out}\ o) & \text{if } x = \mathsf{out}\ o. \end{cases}$$

Then $\tau' \, s_0 \, []$ computes F. □

B.2 Stalling Interrogations

We here give the left out proofs that stalling interrogations as described in Section 9 and interrogations are equivalent.

Lemma 39. *Functionals computable via extended interrogations are computable via stalling interrogations.*

Proof. Let F be computable using a type S and element s_0 by τ via extended interrogations. We use the same type S and element s_0 and define τ' to never use stalling:

$$\tau' \, i \, s \, l := \tau \, i \, s \, l \gg\!= \lambda x. \begin{cases} \mathsf{ret}\ (\mathsf{ask}\ (s', \mathsf{Some}\ q)) & \text{if } x = \mathsf{ask}\ (s', q) \\ \mathsf{ret}\ (\mathsf{out}\ o) & \text{if } x = \mathsf{out}\ o. \end{cases}$$

Then τ' computes F via stalling interrogations. □

Lemma 40. *Functionals computable via stalling interrogations are computable via extended interrogations.*

Proof. Take $\tau \colon I \to S \to A^* \rightharpoonup (S \times Q^?) + O$ computing F via stalling interrogations. We construct $\tau' \, i \, s \, as$ to iterate the function $\lambda s'. \; \tau \, i \, s' \, as$ of type $S \rightharpoonup (S \times Q^?) + O$. If $\mathsf{ask}\ (s'', \mathsf{None})$ is returned, the iteration continues with s''. If $\mathsf{ask}\ (s, \mathsf{Some}\ q)$ is returned, $\tau' \, i \, s as$ returns $\mathsf{ask}\ (s, q)$. If $\mathsf{out}\ o$ is returned, $\tau' \, i \, s \, as$ returns $\mathsf{out}\ o$ as well.

We omit the technical details how to implement this iteration process using unbounded search $\mu : (\mathbb{N} \rightharpoonup \mathbb{B}) \rightharpoonup \mathbb{N}$. □

B.3 Proofs of Closure Properties

We here give the proofs that executing two computable functionals one after the other, composing computable functionals, and performing an unbounded search on a computable functional are all computable operations as stated in Section 4. We explain the tree constructions, which are always the core of the argument. The verification of the trees are then tedious but relatively straightforward inductions, we refer to the Coq code for full detail.

Proof (of Lemma 7). Let τ_1 compute F_1 maping relations $R\colon Q \to A \to \mathbb{P}$ to relations of type $I \to O' \to \mathbb{P}$, and τ_2 compute F_2 mapping relations $R\colon Q \to A \to \mathbb{P}$ to relations of type $(I \times O') \to O \to \mathbb{P}$.

To compute the functional mapping an oracle $R\colon Q \to A \to \mathbb{P}$ to a computation $\lambda io.\, \exists o'\colon O'.\, F_1\, R\, i\, o' \land F_2\, R\, (i, o')\, o$ of type $I \to O \to \mathbb{P}$ we construct a stalling tree with state type $(O' \times \mathbb{N})^?$ and starting state None. The intuition is that the state s remains None as long as τ_1 asks questions, and once an output o' is produced we save it and the number of questions that were asked until then in the state, which remains unchanged after. Then, τ_2 can ask questions, but since as contains also answers to questions of τ_1, we drop the first n before passing it to τ_2.

Formally, the tree takes as arguments the input i, state s ans answer list as, and returns

$$\begin{cases} \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{None}, \mathsf{Some}\ q)) & \text{if } s = \mathsf{None}, \tau_1\, i\, as \rhd \mathsf{Some}\ (\mathsf{ask}\ q) \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{Some}\ (o', |as|), \mathsf{None})) & \text{if } s = \mathsf{None}, \tau_1\, i\, as \rhd \mathsf{Some}\ (\mathsf{out}\ o') \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{Some}\ (o', n), \mathsf{Some}\ q)) & \text{if } s = \mathsf{Some}\ (o', n), \tau_2\, (i, o')\, (as \uparrow_n) \rhd \mathsf{Some}\ (\mathsf{ask}\ q) \\ \mathsf{ret}\ (\mathsf{ask}\ (\mathsf{Some}\ (o', n), \mathsf{Some}\ q)) & \text{if } s = \mathsf{Some}\ (o', n), \tau_2\, (i, o')\, (as \uparrow_n) \rhd \mathsf{Some}\ (\mathsf{out}\ o) \end{cases}$$

where $as \uparrow_n$ drops the first n elements of as. Note that formally, we use bind to analyse the values of τ_1 and τ_2, but just write a case analysis on paper. □

Proof (of Lemma 8). Let τ_1 compute F_1 mapping relations $R\colon Q \to A \to \mathbb{P}$ to relations $X \to Y \to \mathbb{P}$, and τ_1 compute F_2 mapping relations $R\colon X \to Y \to \mathbb{P}$ to relations $I \to O \to \mathbb{P}$. We construct a stalling tree τ computing a functional mapping $R\colon Q \to A \to \mathbb{P}$ to $\lambda io.\ F_2\, (F_1 R)\, i\, o$ of type $I \to O \to \mathbb{P}$.

Intuitively, we want to execute τ_2. Whenever it asks a question x, we record it and execute $\tau_1\, x$ to produce an answer. Since the answer list as at any point will also contain answers of the oracle produces for any earlier question x' of τ_2, we record furthermore how many questions were already asked to the oracle to compute $\tau_1 x$.

As state type, we thus use $(X \times Y)^* \times (X \times \mathbb{N})^?$, where the first component remembers questions and answers for τ_2, and the second component indicates whether we are currently executing τ_2 (then it is None), or τ_1, when it is $\mathsf{Some}\ (x, n)$ to indicate that on answer list as we need to run $\tau_1\, x\, (as \downarrow^n)$, where $as \downarrow^n$ contains the last n elements of as. The initial state is $([\,], \mathsf{None})$.

We define τ to take as arguments an input i, a state (t, z), and an answer list as and return

$$\begin{cases} \mathsf{out}\ o & \text{if } x = \mathsf{None}, \tau_2\, i\, (\mathsf{map}\, \pi_2\, t) \rhd \mathsf{out}\ o \\ \mathsf{ask}\ (t, \mathsf{Some}\ (x, 0), \mathsf{None}) & \text{if } x = \mathsf{None}, \tau_2\, i\, (\mathsf{map}\, \pi_2\, t) \rhd \mathsf{ask}\ x \\ \mathsf{ask}\ (t, \mathsf{Some}\ (x, \mathsf{S}\ n), \mathsf{Some}\ q) & \text{if } x = \mathsf{Some}\ (x, n), \tau_1\, x\, (as \uparrow^n) \rhd \mathsf{ask}\ q \\ \mathsf{ask}\ (t +\!\!+ [(x, y)], \mathsf{None}, \mathsf{None}) & \text{if } x = \mathsf{Some}\ (x, n), \tau_1\, x\, (as \uparrow^n) \rhd \mathsf{out}\ y \end{cases}$$

Intuitively, when we are in the mode to execute τ_2 and it returns an output, we return the output. If it returns a question x, we change mode and stall. When we are in the mode to execute τ_1 to produce an answer for x, taking the last n given answers into account and it asks a question q, we ask the question and indicate that now one more answer needs to be taken into account. If it returns an output y, we add the pair $[(x, y)]$ to the question answer list for τ_1, change the mode back to execute τ_2, and stall. □

Proof (of Lemma 9). We define a tree τ computing the functional mapping $R\colon (I \times \mathbb{N}) \to \mathbb{B} \to \mathbb{P}$ to the following relation of type $I \to \mathbb{N} \to \mathbb{P}$: $\lambda in.\ R\,(i, n)\,\mathsf{true} \land \forall m < n.\ R\,(i, m)\,\mathsf{false}$.

$$\tau\ i\ as := \begin{cases} \mathsf{ret}\ (\mathsf{out}\ i) & \text{if } as[i] = \mathsf{true} \\ \mathsf{ret}\ (\mathsf{ask}\ (i, |as|)) & \text{if } \forall j.\ as[j] = \mathsf{false} \end{cases}$$

Note that a function $\mathsf{find}\ as$ computing the smallest i such that as at position i is true, and else returning None is easy to implement.

Intuitively, we just ask all natural numbers as questions in order. On answer list l with length n, this means we have asked $[0, \dots, n-1]$. We check whether for one of these the oracle returned true, and else ask $n = |l|$. □

C Relation to Bauer's Turing Reducibility

We show the equivalence of the modulus continuity as defined in Lemma 1 with the order-theoretic characterisation used by Bauer [3]. The latter notion is more sensible for functionals acting on functional relations, so we fix some

$$F : (Q \rightsquigarrow A) \to (I \rightsquigarrow O)$$

where $X \rightsquigarrow Y$ denotes the type of functional relations $X \to Y \to \mathbb{P}$. To simplify proofs and notation, we assume extensionality in the form that we impose $R = R'$ for all $R, R' : X \rightsquigarrow Y$ with $Rxy \leftrightarrow R'xy$ for all $x : X$ and $y : Y$.

To clarify potential confusion upfront, note that Bauer does not represent oracles on $\mathbb{N}$ as (functional) relations but as pairs (X, Y) of disjoint sets with $X, Y : \mathbb{N} \to \mathbb{P}$, so his oracle computation operate on such pairs. However, since such a pair (X, Y) gives rise to a functional relation $R : \mathbb{N} \rightsquigarrow \mathbb{B}$ by setting $R\,n\,b := (X\,n \land b = \mathsf{true}) \lor (Y\,n \land b = \mathsf{false})$ and, conversely, $R : \mathbb{N} \rightsquigarrow \mathbb{B}$ induces a pair (X, Y) via $X\,n := R\,n\,\mathsf{true}$ and $Y\,n := R\,n\,\mathsf{false}$, Bauer's oracle functionals correspond to our specific case of functionals $(\mathbb{N} \rightsquigarrow \mathbb{B}) \to (\mathbb{N} \rightsquigarrow \mathbb{B})$. He then describes the computable behaviour of an oracle functional by imposing continuity and a computational core operating on disjoint pairs (X, Y) of enumerable sets that the original oracle functional factors through, which in our chosen approach correspond to the existence of computation trees. So while the overall setup of our approach still fits to Bauer's suggestion, we now show that our notion of continuity is strictly stronger than his by showing the latter equivalent to modulus continuity.

Informally, Bauer's notion of continuity requires that F preserves suprema, which given a non-empty directed set : $(Q \rightsquigarrow A) \to \mathbb{P}$ of functional relations requires that $F\,(\bigcup_{R \in S} R) = \bigcup_{R \in S} F\,R$, i.e. that the F applied to the union of S should be the union of F applied to each R in S. Here directedness of S means that for every $R_1, R_2 \in S$ there is also $R_3 \in S$ with $R_1, R_2 \subseteq R_3$, which ensures that the functional relations included in S are compatible so that the union of S is again a functional relation.

Lemma 41. *If F is modulus-continuous, then it preserves suprema.*

Proof. First, we observe that F is monotone, given that from $F\,R\,i\,o$ we obtain some modulus $L : Q^*$ that directly induces $F\,R'\,i\,o$ for every R' with $R \subseteq R'$.

So now S be directed and non-empty, we show both inclusions separately. First $\bigcup_{R\in S} F\,R \subseteq F\,(\bigcup_{R\in S} R)$ follows directly from monotonicity, since if $F\,R\,i\,o$ for some $R \in S$ we also have $F\,(\bigcup_{R\in S} R)\,i\,o$ given $R \subseteq \bigcup_{R\in S} R$.

Finally assuming $F\,(\bigcup_{R\in S} R)\,i\,o$, let $L : Q^*$ be a corresponding modulus, so in particular $L \subseteq \mathsf{dom}(\bigcup_{R\in S} R)$. Using directedness (and since S is non-empty), by induction on L we can find $R_L \in S$ such that already $L \subseteq \mathsf{dom}(R_L)$. But then also $F\,R_L\,i\,o$ since L is a modulus and R_L agrees with $\bigcup_{R\in S} R)$ on L. □

Lemma 42. *If F is preserves suprema, then it is modulous continuous.*

Proof. Again, we first observe that F is monotone, given that for $R \subseteq R'$ the (non-empty) set $S := \{R, R'\}$ is directed and hence if $F\,R\,i\,o$ we obtain $F\,R'\,i\,o$ since $R' = \bigcup_{R\in S} R$.

Now assuming $F\,R\,i\,o$ we want to find a corresponding modulus. Consider

$$S := \{R_L \mid L \subseteq \mathsf{dom}(R)\}$$

where $R_L\,q\,a := q \in L \wedge R\,q\,a$, so S contains all terminating finite subrelations of R. So by construction, we have $R = \bigcup_{R\in S} R$ and hence $F\,(\bigcup_{R\in S} R)\,i\,o$, thus since F preserves suprema we obtain $L \subseteq \mathsf{dom}(R)$ such that already $F\,R_L\,i\,o$. The remaining part of L being a modulus for $F\,R\,i\,o$ follows from monotonicity. □