Continuous and Monotone Machines

Michal Konečný
School of Engineering and Applied Science, Aston University, Birmingham, UK
m.konecny@aston.ac.uk

Florian Steinberg
INRIA Saclay Île-de-France, Palaiseau, France
fsteinberg@gmail.com

Holger Thies
Department of Informatics, Kyushu University, Japan
thies@inf.kyushu-u.ac.jp

Abstract

We investigate a variant of the fuel-based approach to modeling diverging computation in type theories and use it to abstractly capture the essence of oracle Turing machines. The resulting objects we call continuous machines. We prove that it is possible to translate back and forth between such machines and names in the standard function encoding used in computable analysis. Put differently, among the operators on Baire space, exactly the partial continuous ones are implementable by continuous machines and the data that such a machine provides is a description of the operator as a sequentially realizable functional. Continuous machines are naturally formulated in type theories and we have formalized our findings in Coq as part of Incone, a Coq library for computable analysis.

The correctness proofs use a classical meta-theory with countable choice. Along the way we formally prove some known results such as the existence of a self-modulating modulus of continuity for partial continuous operators on Baire space. To illustrate their versatility we use continuous machines to specify some algorithms that operate on objects that cannot be fully described by finite means, such as real numbers and functions. We present particularly simple algorithms for finding the multiplicative inverse of a real number and for composition of partial continuous operators on Baire space. Some of the simplicity is achieved by utilizing the fact that continuous machines are compatible with multivalued semantics.

2012 ACM Subject Classification Theory of computation → Logic and verification; Mathematics of computing → Continuous mathematics

Keywords and phrases Computable Analysis, exact real computation, formal proofs, proof assistant, Coq

Digital Object Identifier 10.4230/LIPIcs.MFCS.2020.56

Related Version Full version available at `https://arxiv.org/abs/2005.01624`.

Supplementary Material Project Page `https://holgerthies.github.io/continuous-machines/`

Funding *Michal Konečný*: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.
Holger Thies: Supported by JSPS KAKENHI Grant Numbers JP18J10407 and JP20K19744 and by the Japan Society for the Promotion of Science (JSPS), Core-to-Core Program (A. Advanced Research Networks).

1 Introduction

The main goal of this paper is to add to the tools available for producing correct and efficient software with strict specifications that involve high-level mathematical concepts. Such methods are required, for example, for reliable simulations of safety-critical physical

systems, and the number of applications is steadily growing. Computable analysis is a formal model for reliable computation involving real numbers and other spaces of interest in analysis. It extends classical computability theory from discrete structures to continuous ones, replacing natural numbers as codes for abstract objects by elements of Baire space. Computable analysis originated with Turing's fundamental work [27] and was later extended to a theory of computation on real numbers and functions [9, 17], and to more general spaces by Kreitz and Weihrauch [16, 29]. The current work arose as part of our effort to contribute to the development of a framework for conveniently formulating algorithms from computable analysis in a setting that is both fully computational and features formal correctness proofs. Our work has meanwhile been used to extend Incone [26], a computable analysis library based on the proof assistant Coq. Formalizing known results may not be particularly creative, but difficulties encountered in such an endeavour often lead to new developments. In the present case, attempts to avoid overly heavy use of Coq's dependent type system, and to maintain executability within Coq in the presence of non-computational axioms have lead us to concepts that we believe to be of theoretical interest.

We introduce "continuous machines" as an encoding of partial continuous operators derived from the fuel-based approach to modeling diverging computation in intuitionistic type theories [1]. Continuous machines can be understood as an abstraction of oracle machines as used to introduce the model of computation central to computable analysis. There are two main points that support this analogy and distinguish our approach from uses in type theory: The first is the presence of a functional parameter that is considered an input and that takes the role of the oracle in an oracle machine. Machines are type-two objects, which is crucial as it makes continuity and information theoretic arguments applicable. The second particularity is a curried discrete input, meaning that for fixed functional input we get a function that we consider the return value if it is total, if it is not total the return value is undefined. As a consequence, the natural domains need not be open but only G_δ sets.

These two features reflect that we really encode continuous operators, i.e. partial functions from Baire space to Baire space, as opposed to partial continuous functionals. We consider operator composition a natural operation, while for functionals the same operation would be called functional substitution and is less important [5]. The emphasis on operators is in tune with the principle of computable analysis to almost consistently replace the natural numbers by Baire space as the base type. The reason this works is that partial continuous operators can be encoded as elements of Baire space via the application of a partial combinatory algebra [12, 15]. A partial operator is computable by an oracle machine if and only if there is a computable code. Computability as a functional is also equivalent [22].

Working directly with codes from Baire space is tedious and our main result, Theorem 6, shows that continuous machines are completely equivalent: It provides a full translation from a continuous machine to a code from Baire space and back that preserves computability. As type-two objects continuous machines are a high-level concept and more convenient for defining partial operators. We illustrate just how concisely algorithms on continuous data can be formulated using continuous machines at the example of inversion of a real number in Section 3.1. As another example we describe a simple and fairly efficient implementation of the composition of partial operators encoded as continuous machines in Section 4.2. We have fully automated the translations between continuous machines and Baire-space codes in that we have defined them in Coq and provide complete formal proofs of correctness. In fact, also the two main examples, all other important points made in this paper, and further examples whose description we omit for space reasons, have been formalized. This should be kept in mind as it justifies cutting down on some details for the sake of communicating the

important ideas. Instructions on how to access our formal development and relate it to the contents of the paper can be found on the project homepage (see supplement material listed on the first page).

The topics of this paper can be viewed from a number of different perspectives. Clearly there are links to type theory. In particular, the results have been formalized in the type theory based proof assistant Coq. The main object of investigation can be understood as a variant of the fuel-based approach to modeling divergent computations in constructive type theories. A survey of such methods presented in type-theoretic language and many relevant references can be found in [1]. However, here we avoid a type-theory-like presentation and prefer mathematical notation consisting of a mix of conventions commonly found in computable analysis and game-centered parts of higher-order computability and programming language theory [21]. In particular we choose to illustrate the use of fuel with Turing machines and oracle machines directly to avoid pointers to type theory in the body of the paper.

The connection of our work to higher-order computability theory is reflected in a possible interpretation of the main result: we provide yet another characterization of the sequentially realizable functionals [20]. The connection to computable analysis is also evident and it is our main source of examples. Some of these examples nicely illustrate connections to precompleteness, constructions forcing precompleteness and completions [16, 24]. These concepts have quite some history but have recently been rediscovered for their applications in the theory of Weihrauch reductions and in complexity theory for computable analysis [7, 2].

Partial operators on Baire space can also be captured in Coq's type theory, where partiality is reflected in the use of sigma-types as inputs, i.e. by taking as input a dependent pair of the actual input and a proof that this input is from its domain. While continuous machines provide additional information over a direct definition in Coq, we believe that on the computability level this difference is irrelevant as the information can be read off from each respective Coq term. For an equivalent of a fragment of the Coq terms, an extraction of the additional information a continuous machine provides has been formalized in Agda [30]. We know that an internal formulation of such a result that covers all definable functions in Coq is impossible. It involves extracting a modulus of continuity and there are known obstructions to extracting this information extensionally [10]. The support of tactics and Coq's formalization of Coq, i.e. the MetaCoq project [25] should allow to adapt the work in Agda and extend the extraction to cover all or at least the majority of relevant Coq terms [8].

We give a quite detailed sketch of the proof of our main theorem (Theorem 6) but for space reasons did not include most of the other proofs. Some of them can be found in the appendix. Note that fully formal versions of all proofs are also contained in our Coq development and that a longer and more exhaustive version of this paper that also contains some additional results is available [14].

2 Computable analysis revisited

Let $\mathbf{Q}$ and $\mathbf{A}$ be two sets that we understand to consist of **questions** and **answers**. We always assume these sets to be countable, and in all concrete examples considered specifying explicit bijections with the natural numbers is straightforward. In the following we restate standard definitions from computable analysis where we insert $\mathbf{Q}$ and $\mathbf{A}$ for the appropriate copies of $\mathbb{N}$. A **representation** of a set X is a partial surjective function $\delta\colon\subseteq \mathbf{A}^{\mathbf{Q}} \to X$. For $x \in X$, each $\varphi\colon \mathbf{Q} \to \mathbf{A}$ with $\delta(\varphi) = x$ is called a **name** of x and should be understood to provide on-demand information about x. I.e. if φ is a name of x then given a question $q \in \mathbf{Q}$ about x the value $\varphi(q) \in \mathbf{A}$ is a valid answer to the question. Call $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ the **space of names** of the representation, $\mathcal{B}$ due to the case $\mathbf{Q} = \mathbb{N} = \mathbf{A}$ where it is the **Baire space**.

A **represented space** is a pair $\mathbf{X} := (X, \delta_{\mathbf{X}})$ where $\delta_{\mathbf{X}}$ is a representation of X. The representation induces topological and computability structures on the set X. Namely, X can be made a topological space by considering the final topology of the representation and an element of a represented space is called **computable** if it has a computable name. The latter of course presumes that $\mathbf{Q}$ and $\mathbf{A}$ are such that it is clear what computability of a function from $\mathbf{Q}$ to $\mathbf{A}$ means; which is in particular the case when $\mathbf{Q}$ and $\mathbf{A}$ come with explicit bijections to $\mathbb{N}$. More generally, $\mathbf{Q}$ and $\mathbf{A}$ can be thought of as being equipped with a numbering. If a topological space is given and a representation is to be constructed, then the candidates are expected to reproduce the given topology.

▶ **Example 1** ($\mathbb{R}_{\mathbb{Q}}$: Reals via rational approximations)**.** One possible way to represent real numbers is via rational approximations. The **rational representation** of the real numbers is the unique partial function $\delta_{\mathbb{R}_{\mathbb{Q}}} :\subseteq \mathbb{Q}^{\mathbb{Q}} \to \mathbb{R}$ that satisfies the specification

$$\delta_{\mathbb{R}_{\mathbb{Q}}}(\varphi) = x \quad \iff \quad \forall \varepsilon > 0 \colon |x - \varphi(\varepsilon)| \leq \varepsilon.$$

We denote the corresponding represented space by $\mathbb{R}_{\mathbb{Q}}$ and use it as one of the running examples. The represented space $\mathbb{R}_{\mathbb{Q}}$ is widely considered to provide the "correct" computability structure on the real numbers and sometimes even used as a benchmark representation in work that reasons about complexity in the setting of computable analysis [13, 18]. The rational representation is fairly convenient: It provides a simple question and answer structure and an intuitive interface for accessing information about real numbers. It only uses a single additional type, namely $\mathbb{Q}$, which has a well-developed theory in Coq's standard library.

2.1 Continuous and computable functions

Fix some represented spaces $\mathbf{X}$ and $\mathbf{X}'$. Let $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ be the space of names of the representation $\delta_{\mathbf{X}}$ of $\mathbf{X}$ and $\mathcal{B}' := \mathbf{A}'^{\mathbf{Q}'}$ that of $\delta_{\mathbf{X}'}$. As $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{Q}'$ and $\mathbf{A}'$ are countable, we may think of them as discrete spaces and it makes sense to talk about continuity of operators $F :\subseteq \mathcal{B} \to \mathcal{B}'$. Such an operator is continuous if its return values are determined by a finite number of values of its input function. That is, if for all $\varphi \in \operatorname{dom}(F)$ and each $q' \in \mathbf{Q}'$ there exists a finite list of questions $\mathbf{q} \in \operatorname{seq} \mathbf{Q}$ such that

$$\forall \psi \in \operatorname{dom}(F) \colon \varphi|_{\mathbf{q}} = \psi|_{\mathbf{q}} \implies F(\varphi)(q') = F(\psi)(q')$$

where $\varphi|_{\mathbf{q}}$ denotes the restriction of φ to $\mathbf{q}$. Since $\mathbf{Q}$ and $\mathbf{Q}'$ are countable, equivalent definitions can be obtained by introducing metric structures on $\mathcal{B}$ and $\mathcal{B}'$ or by requiring a continuous function to preserve limits of sequences. These equivalences are useful for abstract reasoning about continuity and formal versions are available in the Incone library [26]. In the case where all question and answer sets coincide with the natural numbers, computability of operators can be introduced by means of oracle machines. An oracle machine is a machine with a marked oracle query and answer states and a marked oracle tape. The run of such a machine on oracle $\varphi \in \mathcal{B}$ is defined as the run of a regular machine but any time the oracle query state is entered, the content q of the oracle tape is replaced by $\varphi(q)$ and the state is changed to the answer state. For some background and more details we point the reader to [11]. It should be kept in mind that the oracle is considered an input to the computation and despite the name and other applications of the same concept with fixed oracle, this makes oracle machines a realistic model of computation.

An operator $F :\subseteq \mathcal{B} \to \mathcal{B}'$ is said to **realize** a function $f \colon \mathbf{X} \to \mathbf{X}'$ if for each name φ of some $x \in \mathbf{X}$ the value $F(\varphi)$ is a name of $f(x) \in \mathbf{X}'$. A function between represented spaces is called **continuous** or **computable** if it has a realizer with that property. In all cases we

are interested in, this notion of continuity coincides with topological continuity w.r.t. the final topology of the representations and the natural topology on the spaces. Without going into detail, this is because we only consider **admissible** representations [24].

Continuity is a prerequisite for computability. The real numbers are connected, discrete spaces are totally disconnected and images of connected sets under continuous functions are connected. For this reason each non-constant function from the reals to the Booleans fails to be computable. This argument covers equality checks, comparisons and other operations that are routinely used and seem indispensable for applications. Often, computability can be recovered by replacing a discrete target space by an appropriate non-discrete finite space.

▶ **Example 2** (Sign function and Kleeneans)**.** The sign function is discontinuous as a function from the reals to a discrete space (e.g. its image as a subspace of the reals). A computable version can be recovered by replacing its three possible values with elements of the following space: For any set $\mathbf{Q}$, denote by $\operatorname{opt}\mathbf{Q}$ be the union of $\mathbf{Q}$ with a new element None and use $\operatorname{Some} q$ for the element of $\operatorname{opt}\mathbf{Q}$ corresponding to $q \in \mathbf{Q}$. Consider the three-point set $\{\text{true}_{\mathbb{K}}, \text{false}_{\mathbb{K}}, \bot_{\mathbb{K}}\}$ and equip it with a representation $\delta_{\mathbb{K}}$ on names of type $\mathbb{N} \to \operatorname{opt}\mathbb{B}$ by

$$\delta_{\mathbb{K}}(\varphi) = \begin{cases} b_{\mathbb{K}} & \text{if } \exists n, \varphi(n) = \operatorname{Some} b \text{ and } \forall m < n, \varphi(m) = \text{None} \\ \bot_{\mathbb{K}} & \text{otherwise.} \end{cases}$$

That is, the constant None sequence is a name of $\bot_{\mathbb{K}}$ and for any other sequence the first element that is not None determines which of $\text{true}_{\mathbb{K}}$ and $\text{false}_{\mathbb{K}}$ is named.

We refer to $\mathbb{K} := (\{\text{true}_{\mathbb{K}}, \text{false}_{\mathbb{K}}, \bot_{\mathbb{K}}\}, \delta_{\mathbb{K}})$ as the **Kleeneans** as it models the behavior of three-valued logics considered by Kleene. Note that the representation is total, i.e. all sequences are valid names, which makes it convenient to define realizers of functions into the space. When defining functions that use the Kleeneans as an argument, it is often more convenient to require names to be monotone in the sense that if the sequence contains true or false, the subsequent values remain the same. The use of this restriction does not change the space, as an arbitrary name can be computably transformed into a monotone name. The sign function as a function from the reals to the Kleeneans can be defined from the Boolean comparison on the reals as

$$\operatorname{sign}_{\mathbb{K}}(x) := \begin{cases} (0 < x)_{\mathbb{K}} & \text{if } x \neq 0 \\ \bot_{\mathbb{K}} & \text{otherwise.} \end{cases}$$

Where the strict inequality could as well have been replaced by non-strict inequality as the case $x = 0$ is treated separately anyway. A continuous realizer of the sign as a function of type $\mathbb{R}_{\mathbb{Q}} \to \mathbb{K}$ can be specified from the Boolean comparisons on the rational numbers as

$$F(\varphi)(n) := \begin{cases} \operatorname{Some}(0 < \varphi(2^{-n})) & \text{if } |\varphi(2^{-n})| > 3 \cdot 2^{-n} \\ \text{None} & \text{otherwise.} \end{cases}$$

As comparison of rational numbers is decidable, this realizer is computable. To verify its correctness note that if φ is a name of 0, then $|\varphi(2^{-n}) - 0| \leq 2^{-n}$ implies that F returns the constant None sequence. If φ is a name of some $x \neq 0$ then there exists some n such that $2^{-(n-2)} < |x|$ and thus $|\varphi(2^{-n})| > 3 \cdot 2^{-n}$ by a use of the reverse triangle inequality. Whenever we are in the first case it follows that as Booleans $0 < \varphi(2^{-n}) = 0 < x$. In combination of these we conclude that F returns a name of the correct value.

The requirement to be greater than $3 \cdot 2^{-n}$ in the definition of F can be replaced by just demanding the same value to be greater or equal 2^{-n} while maintaining correctness. However, the former forces that the realizer always returns names that are monotone in the sense discussed above. To verify this note that as $|\varphi(2^{-n}) - \varphi(2^{-(n+1)})| \leq 3 \cdot 2^{-(n+1)}$, whenever $|\varphi(2^{-n})| > 3 \cdot 2^{-n}$ it follows that $|\varphi(2^{-(n+1)})| > 3 \cdot 2^{-(n+1)}$.

3 Multifunctions and abstract machines

In computable analysis it is often the case that continuity fails for extensionality reasons and dropping extensionality by using multivalued functions is a popular and powerful tool to work around such problems. A multivalued function from a set X to another set Y assigns to each element $x \in X$ a set of eligible return values $F(x) \subseteq Y$. This set may be empty and those x for which it is non-empty are considered to constitute the **domain** $\mathrm{dom}(F) \subseteq X$. The multifunction is called **total** if its domain is all of X and **single-valued** if it only returns sub-singletons, i.e. each value set has at most one element. Each partial function can be considered a single-valued multifunction; this multifunction uniquely specifies the partial function and is total if and only if the partial function is.

A partial function f is said to **choose through** a multifunction F if for each $x \in \mathrm{dom}(F)$ it returns an eligible return value, i.e. $f(x)$ is defined and an element of $F(x)$. Note that the domain of the partial function can be bigger than that of the multifunction. A multifunction should be considered a specification of all partial functions that choose through it. This defines an important ordering on the multifunctions: A multifunction F is said to **tighten** another multifunction G, in symbols $F \prec G$, if any partial function that is a choice for F is also a choice for G or equivalently $F \prec G$ if and only if

$$\mathrm{dom}(G) \subseteq \mathrm{dom}(F) \quad \text{and} \quad \forall x \in \mathrm{dom}(G), F(x) \subseteq G(x).$$

For partial functions $f \prec g$ if and only if f is an extension of g. A partial function f chooses through a multivalued F if and only if $f \prec F$. Multivalued functions from X to Y are in one to one correspondence with relations, i.e. subsets of $X \times Y$. However, multivalued functions should be understood as directed and thus the natural operations such as composition differ.

Recall that a function $f\colon \mathbf{X} \to \mathbf{Y}$ between represented spaces is realized by some $F\colon\subseteq \mathcal{B} \to \mathcal{B}$ if F translates names of $x \in \mathbf{X}$ to names of $f(x)$. An alternate way to express this is that $\delta_{\mathbf{Y}} \circ F$ is an extension of $f \circ \delta_{\mathbf{X}}$. This suggests a lift to multivalued functions: a multifunction $g\colon \mathbf{X} \rightrightarrows \mathbf{Y}$ is realized by another multifunction $G\colon \mathcal{B} \rightrightarrows \mathcal{B}$ if $\delta_{\mathbf{Y}} \circ G \prec g \circ \delta_{\mathbf{X}}$. This definition behaves as expected, in particular for partial functions: An operator realizes a partial function if and only if it is a realizer w.r.t. the subspace representation on the argument space. For multifunctions, being a realizer is preserved under tightening the realizer and "loosening" the realized function. As continuity and computability are preserved under composition and multifunction composition is compatible with tightenings, the notions of continuous and computable realizability compose well. While any multivalued function is uniquely determined by its partial choice functions, the same need not be true for continuous partial functions: A continuously realizable multifunction need not have any partial continuous choice functions at all.

As we are mostly interested in continuous and computable realizers, one may argue that allowing multivalued realizers is not necessary. Continuity makes sense only for functions, or at least is known to be problematic in the presence of multivaluedness [23]. However, we shall use a notion of algorithms that can *a priori* give multivalued results. Although it is possible to force single-valuedness, it can be convenient to not always do this right away and the notion of multivalued realizers turns out to be useful. Another consequence that is useful in other parts of computable analysis is that when multivalued realizers are allowed, any multifunction g between represented spaces has a unique realizer that is maximal with respect to tightenings, namely $\delta_{\mathbf{Y}}^{-1} \circ g \circ \delta_{\mathbf{X}}$ [3].

3.1 Algorithmic content and machines

Now that we discussed the tools we need for specification, the next step is to see how to produce computational objects that can fulfill these specifications. In particular, we are interested in devising operators, that is, partial functions on Baire space or Baire-space-like spaces of functions. We take the fuel-based approach for capturing divergence in type theories and adapt it to the operator and oracle machine setting of computable analysis. As before, fix some countable sets $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{Q}'$ and $\mathbf{A}'$ and abbreviate $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ and $\mathcal{B}' := \mathbf{A}'^{\mathbf{Q}'}$. To each function $M\colon \mathcal{B} \to \operatorname{opt}(\mathbf{A}')^{\mathbb{N}\times\mathbf{Q}'}$ assign a multifunction $F_M\colon \mathcal{B} \rightrightarrows \mathcal{B}'$ whose return value on input φ is the set $\{\psi \in \mathcal{B}' \mid \forall q', \exists n, M(\varphi)(n,q') = \operatorname{Some}(\psi(q'))\}$. This set can be empty or contain more than one element but for each $\varphi \in \mathcal{B}$ the set $F_M(\varphi)$ is a closed subset of $\mathcal{B}'$.

Whenever an operator F can be computed by an oracle machine, M can be chosen to be the function that on inputs φ, n and q' runs the oracle machine for up to n time steps on input q' and oracle φ, in case of termination returns $\operatorname{Some} a'$ where a' is what the machine returned and otherwise returns None. Then F_M is single-valued and the corresponding partial function extends F. The values of F can be recovered from those of M by searching through increasing values of n, and for a general M this gives a choice function of F_M. Coq's standard library proves a restricted choice principle called constructive epsilon that can be used to recover a choice function of F_M as a dependently typed function when given an arbitrary function M. Internally this leads to a linear search through the values.

Motivated by the oracle machine example, we call a function M a **machine** for F if F_M tightens F. This analogy should be taken with a grain of salt, and, in particular, it does not appropriately reflect the role played by the natural number input n. We refer to n as the **effort parameter**, and while higher values do usually indicate a higher time consumption of the computation, it need not be directly related to the running time. In particular, we refrain from interpreting the effort parameter as ordering a computation into a sequence of steps and instead embrace the view that it is a functional input.

For illustration, let us discuss the task of finding a multiplicative inverse in the rational representation. Consider $x \mapsto 1/x$ as a partial function on the represented space $\mathbb{R}_{\mathbb{Q}}$ from Example 1. The function is partial as it is undefined in 0. We define a function $M\colon \mathcal{B} \to \operatorname{opt}\mathbb{Q}^{\mathbb{N}\times\mathbb{Q}}$ of which we claim that $F_M\colon \mathcal{B} \rightrightarrows \mathcal{B}$ is a realizer of inversion: Let

$$M(\varphi)(n,\varepsilon) = \begin{cases} \operatorname{Some} \dfrac{1}{\varphi(\min\{\delta, \varepsilon\delta^2\}/2)} & \text{if } \delta := |\varphi(2^{-n})| - 2^{-n} > 0 \\ \operatorname{None} & \text{otherwise.} \end{cases}$$

Or in words: Use n as a guess for how precise one has to know x to bound it away from zero and return an approximation to the inverse that can be straight forwardly computed once x is bounded away from zero. Unfolding of the definitions reveals that to demonstrate the correctness of our assertion we have to prove that for all $x \neq 0$

$$\delta_{\mathbb{R}_{\mathbb{Q}}}^{-1}(x) \subseteq \operatorname{dom}(F_M) \wedge \forall \varphi \in \delta_{\mathbb{R}_{\mathbb{Q}}}^{-1}(x)\colon F_M(\varphi) \subseteq \delta_{\mathbb{R}_{\mathbb{Q}}}^{-1}(1/x).$$

This should be understood as two statements: Firstly that the domain of F_M includes all names of real numbers from the domain of the inversion function, and secondly that on valid arguments it only returns correct values.

To prove the first of these statements, let φ be a name of some $x \neq 0$. It suffices to pick n larger than $\log_2(|1/x|)$ to avoid the second case and thus the domain of F_M is big enough. To check the second condition, i.e. that F_M only returns correct values, let φ be a name of x. It suffices to check for each $\varepsilon > 0$ that $M(\varphi)(n,\varepsilon) = \operatorname{Some} r$ implies that $|r - 1/x| \leq \varepsilon$. If the assumption of this implication is true, then we have $\delta := |\varphi(2^{-n})| - 2^{-n} > 0$ and we know

the value of r. First note that $|x| \geq \delta$ as can be seen using the inverse triangle inequality and that δ is positive. This, together with another application of the inverse triangle inequality, leads to

$$|\varphi(\min\{\delta, \varepsilon\delta^2\}/2)| \geq \left||x| - \frac{\min\{\delta, \varepsilon\delta^2\}}{2}\right| \geq \frac{\delta}{2}$$

and allows us to conclude that

$$\begin{aligned}\left|\frac{1}{\varphi(\min\{\delta, \varepsilon\delta^2\}/2)} - \frac{1}{x}\right| &= \frac{|\varphi(\min\{\delta, \varepsilon\delta^2\}/2) - x|}{|\varphi(\min\{\delta, \varepsilon\delta^2\}/2)x|} \\ &\leq \frac{\min\{\delta, \varepsilon\delta^2\}}{\delta^2} \leq \varepsilon.\end{aligned}$$

As the left-hand side is the value of r this proves the correctness of return values.

The function M is computable as all operations it uses on the rational numbers are computable. Note that F_M is properly multivalued. In general, computability should imply continuity, but this does not make sense here as continuity only makes sense for single-valued functions. This can be resolved by removing the multivaluedness of F_M via picking its value on the smallest n for which it returns something. Realizability is preserved under tightening, thus the function obtained in this way is a realizer again. As searching is a computable operation, this realizer is still computable, which is reflected in the fact that it can be defined as a dependently typed function in Coq from the definition of M as above.

4 Machines as names of functions

It is true that for every partial operator F there exists some machine M such that F_M extends F. This means that we can understand M as a description of F in a similar way to how representations work. Nevertheless, as the candidates for question and answer sets are full function spaces and thus uncountable, this does not formally define a representation. Access to M alone is an inconvenient set of information in the sense that it is difficult to maintain. For instance, given such information for two operators F and G, it can not be easily found for the operator $F \circ G$. This is because $G(\varphi)$ and thus the input for F can only be approximated from access to a machine for G. Only when restricting to continuous operators, can one hope to succeed by extending some finite sub-function in an arbitrary way. To guarantee that this does not interfere with the correctness of the return values, one needs explicit continuity information about F.

Fix some $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{Q}'$ and $\mathbf{A}'$ and use the abbreviations $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ and $\mathcal{B}' := \mathbf{A}'^{\mathbf{Q}'}$. A set of continuity information about a continuous function that is often used in constructive analysis is a modulus function. A function $\mu \colon \subseteq \mathcal{B} \to (\operatorname{seq} \mathbf{Q})^{\mathbf{Q}'}$ is called a **modulus** of an operator $F \colon \subseteq \mathcal{B} \to \mathcal{B}'$ if it is a Skolem function of the continuity statement from Section 2.1 in the sense that for all $\varphi, \psi \in \operatorname{dom}(F)$ and $q' \in \mathbf{Q}'$

$$\varphi|_{\mu(\varphi)(q')} = \psi|_{\mu(\varphi)(q')} \implies F(\varphi)(q') = F(\psi)(q'). \tag{1}$$

In particular $\operatorname{dom}(F) \subseteq \operatorname{dom}(\mu)$ as otherwise the premise of the implication does not make sense. A modulus μ may itself be considered an operator and the type of a modulus of μ coincides with the type of μ itself. It thus makes sense to call a modulus **self-modulating** if it is its own modulus.

▶ **Proposition 3.** *Any continuous partial operator has a self-modulating modulus of continuity.*

Call a pair (M,μ) a **continuous machine** if M is of type $\mathcal{B} \to \operatorname{opt} \mathbf{A}'^{\mathbb{N}\times\mathbf{Q}'}$ and μ is a self-modulating modulus of M. Say that a continuous machine (M,μ) implements an operator $F\colon\subseteq \mathcal{B} \to \mathcal{B}'$ if F_M tightens F. Let us emphasize that the function μ above is a modulus of continuity of the machine M itself and not of a potential operator F that it computes. In particular, just like M itself, the modulus μ is always a total function. Proposition 5 below implies that from μ one can obtain a modulus of continuity of any operator that is tightened by F_M. However, it is not difficult to construct a discontinuous M such that F_M is a continuous partial function and thus a modulus of F_M is not enough information to recover one of M. Thus, a modulus of the computed operator should be considered to provide strictly less information than μ.

If M is constructed from an oracle machine, a computable self-modulating modulus μ for M can be readily read off the oracle machine by following the queries that the machine writes to its oracle tape. The resulting pair (M,μ) is a continuous machine that implements F. More generally, every continuous operator can be implemented by a continuous machine.

▶ **Proposition 4.** *If $F\colon\subseteq \mathcal{B} \to \mathcal{B}'$ is continuous then there exists some continuous machine that implements it.*

Just like it is possible to reconstruct the values of F from M, a modulus for F can be reconstructed using the additional information that a continuous machine provides.

▶ **Proposition 5.** *A machine that computes a modulus for $F\colon\subseteq \mathcal{B} \to \mathcal{B}$ can be obtained in a fully uniform way from a continuous machine that implements F. The construction can be done in such a way that it preserves being self-modulating.*

Here and in the following results by "fully uniformly" we mean that the transformation can be defined in a fragment of Coq's type theory small enough to not go beyond definability in system T when all the question and answer types are the natural numbers. Adding a self-modulating modulus μ to a machine completes the set of information about F in the sense that a continuous machine implementing a realizer of some function between represented spaces contains exactly the amount of information that one would expect to be specified about such a function in computable analysis. To understand this in more detail let us first recall how computable analysis treats spaces of functions.

4.1 Function spaces and continuous machines

In this part we make the additional assumption that the question type $\mathbf{Q}$ features decidable equality to allow encoding finite functions as lists over $\operatorname{seq}(\mathbf{Q}\times\mathbf{A})$. The decidable equality on $\mathbf{Q}$ is needed to make evaluation and checking for inclusion of a finite list in the domain of the finite function definable. In Incone each question type of a represented space comes with a default question $q_d \in \mathbf{Q}$. For convenience this section also assumes a default answer to be available. This can be avoided by using the answer for the default question instead.

Fix some represented spaces $\mathbf{X}$ and $\mathbf{X}'$ whose spaces of names are $\mathcal{B} = \mathbf{A}^{\mathbf{Q}}$ and $\mathcal{B}' = \mathbf{A}'^{\mathbf{Q}'}$ respectively. In the following use ? as notation for the right inclusion into $\mathbf{A}' + \operatorname{seq}\mathbf{Q}$ and ! for the left inclusion, i.e. a question mark for a list of questions and an exclamation mark for an answer. For a fixed function $\psi\colon \operatorname{seq}(\mathbf{Q}\times\mathbf{A})\times\mathbf{Q}' \to \mathbf{A}' + \operatorname{seq}\mathbf{Q}$ and fixed $\varphi \in \mathcal{B}$ and $q' \in \mathbf{Q}'$ inductively define a sequence of finite functions $\phi_n \in \operatorname{seq}(\mathbf{Q}\times\mathbf{A})$ by $\phi_0 := \epsilon$ and

$$\phi_{n+1} := \begin{cases} \phi_n +\!\!+ \varphi|_{\mathbf{q}} & \text{if } \psi(\phi_n, q') = ?\mathbf{q} \\ \phi_n & \text{otherwise.} \end{cases}$$

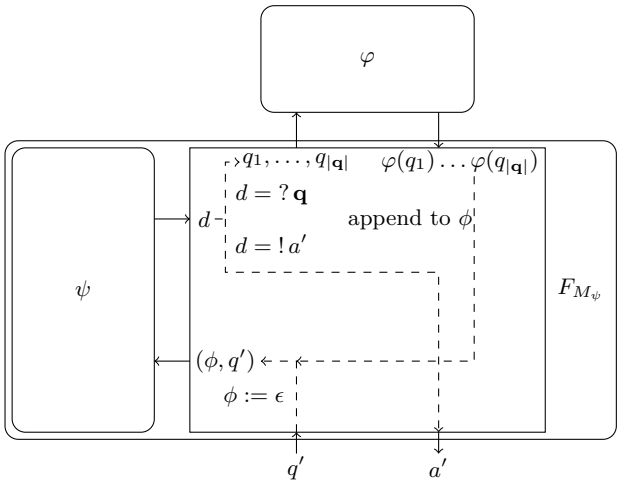

Figure 1 Some ψ is a name of a function $f : \mathbf{X} \to \mathbf{X}'$ iff F_{M_ψ} realizes f. The box with pointy corners represents a realistically implementable algorithm while φ and ψ may be computable or non-computable and are therefore depicted with rounded corners. Whenever ψ is computable and has pointy corners, also F_{M_ψ} will be computable and can be depicted with pointy corners.

From this sequence define a machine M_ψ as follows:

$$M_\psi(\varphi)(n, q') := \begin{cases} \text{Some}\, a' & \text{if } \psi(\phi_n, q') = !a' \\ \text{None} & \text{otherwise.} \end{cases}$$

The function-space representation $\delta_{\mathbf{X}'\mathbf{X}}$ assigns ψ as name to a function $f\colon \mathbf{X} \to \mathbf{X}'$ if and only if F_{M_ψ} realizes f (see Figure 1). That is, the space of names is given by $\mathcal{B}_{\mathbf{X}'\mathbf{X}} = (\mathbf{A}' + \operatorname{seq} \mathbf{Q})^{\operatorname{seq}(\mathbf{Q}\times\mathbf{A})\times\mathbf{Q}'}$. In particular the questions and answers are countable. While our presentation is different, the central idea coincides with that behind Weihrauch's η [29]. Straightforward computations show that $\mu_\psi(\varphi)(n, q') := \operatorname{dom}(\phi_n)$ is a self-modulating modulus of M_ψ and thus (M_ψ, μ_ψ) is a continuous machine and that F_{M_ψ} is single-valued and therefore also continuous by Proposition 5. The set underlying the represented space $\mathbf{X}^{\mathbf{X}'}$ are exactly the continuous functions. Call ψ an **associate** of $F\colon \mathcal{B} \rightrightarrows \mathcal{B}'$ if F_{M_ψ} tightens F. Then ψ is a name of f if and only if it is an associate of a realizer of f and any associate of F can be used to obtain a continuous machine that implements F.

Let us argue that the converse also holds, i.e. that from a continuous machine (M, μ) one can obtain an associate $\psi_{M,\mu}$ of F_M. To get an intuition for what an associate of F_M should be doing, first consider the computable case where an actual oracle machine is available. The main obstacle in this case is that the associate is required to be total and divergences of the oracle machine need to be taken care of. Define an associate ψ of the operator as follows: given a finite function ϕ and some question q' run the oracle machine for at most $|\phi|$ steps while looking up the answers to the questions that the oracle machine asks in the finite function. If the lookup fails for a question $q \in \mathbf{Q}$, then return $?(q)$. If all lookups are successful and the machine terminates with return value a' return $!a'$. In case that $|\phi|$ steps are exceeded without either happening, return $?(q_d)$ where q_d is the default question of $\mathbf{X}$.

Next, let us discuss how to supplement full inspection capabilities into how an operator is computed with access to a self-modulating modulus. For illustration, we consider only the special case where F is total and let μ be a self-modulating modulus of F itself. That is, we drop the effort parameter and remark that this simplification is partially justified by the last

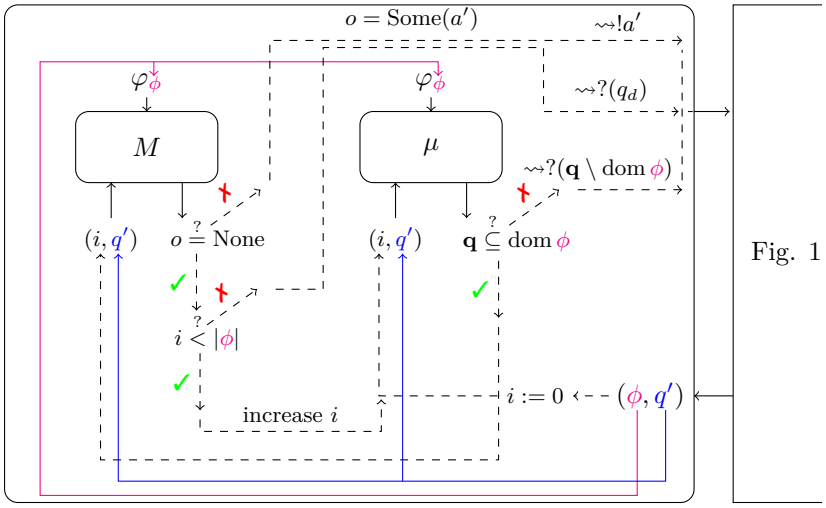

Fig. 1

Figure 2 Constructing an associate $\psi_{M,\mu}$ from a continuous machine (M, μ). Here, φ_ϕ is the total function that extends ϕ with a default value. If M and μ come with algorithms to compute them, we obtain an algorithm for $\psi_{M,\mu}$.

paragraph, which argues that divergences can be taken care of. Thus, for fixed inputs φ, q', an associate should attempt to get hold of $\mu(\varphi)(q')$, as this is the set of arguments whose return-values its final answer should depend on. However, the associate does not have access to φ but only to a finite sub-function ϕ. Let φ_ϕ denote the constant extension of ϕ with some default answer a_d. The associate may on input of ϕ and q' use φ_ϕ as a replacement for φ. However, φ and φ_ϕ can only be expected to coincide on $\mathrm{dom}(\phi)$ and so $\mu(\varphi)(q')$ and $\mu(\varphi_\phi)(q')$ could be different. This is where μ being self-modulating comes in: the values of the modulus coincide whenever $\mu(\varphi_\phi)(q') \subseteq \mathrm{dom}(\phi)$, and this condition can be checked by the associate. Thus, let the associate on input of ϕ and q' check whether $\mu(\varphi_\phi)(q') \subseteq \mathrm{dom}(\phi)$. If this test fails ask for the difference, i.e. return $?(\mu(\varphi_\phi)(q') \setminus \mathrm{dom}(\phi))$. If the test is successful, then $\mu(\varphi_\phi)(q') = \mu(\varphi)(q')$ and the associate can safely return $!F(\varphi_\phi)(q')$ as μ is a modulus of F.

By construction, the candidate for an associate that we implicitly defined in the last paragraph is an associate of a restriction of F. However, as the modulus is evaluated on functional inputs different from the actual input in relevant places, an argument is needed to see that the iteration is always finite. First argue that the sequence $\varphi_n := \varphi_{\phi_n}$ converges to some limit $\psi \in \mathcal{B}_{\mathbf{X}}$. This is because for some fixed $q \in \mathbf{Q}$ either there exists some n such that $q \in \mathrm{dom}(\phi_n)$, in which case $\varphi_k(q) = \varphi(q)$ for all k bigger than n, or there does not exist such an n and $\varphi_k(q) = a_d$ for all k. As μ is self-modulating, it is continuous, and since all question and answer types are countable, also sequentially continuous. Thus $\mu(\varphi_n)$ converges to $\mu(\psi)$ and there exists some k such that $\mu(\varphi_m)(q') = \mu(\psi)(q')$ for any $m \geq k$. In particular $k + 1$ is a sufficiently large effort to lead the evaluation of the associate to return a value. Figure 2 shows the behaviour of the associate we used for the proof of the following theorem.

▶ **Theorem 6.** *There exists a fully uniform way to construct from a continuous machine (M, μ) and default elements $q_d \in \mathbf{Q}$ and $a_d \in \mathbf{A}$ an associate of F_M.*

4.2 Continuous machines and monotonicity

Continuous machines and associates theoretically contain the same information about a continuous operator. However, in practice continuous machines are vastly superior to associates if the task is to directly implement and formally prove the correctness of an

algorithm. The skeptical reader may revisit the example of inversion from Section 3.1, supplement a self-modulating modulus and extract an associate. Totality of the associate and encoding finite functions by lists introduces irrelevant default values and correctness proofs tedious due to complicated induction recursion arguments. Translating from continuous machines takes part of these burdens off the user.

As the concept of an associate is linked to partial combinatory algebras (c.f. for instance [28]), many operations on continuous operators are in principle implementable for associates and thus also for continuous machines. For implementation of operations on continuous operators, both working with associates and working with machines is unhandy but for somewhat different reasons. While associates are difficult to construct, a continuous machine as input makes some important information not readily available. One way of reflecting the difference in rigidity of the concepts is to translate back and forth between them. While any continuous machine can be translated to an associate, the machines that are obtained from an associate have very special properties some of which can be maintained separately. Strictly speaking, that the modulus is self-modulating is already an example of such a property as, so far, all arguments still work if it is only required to be sequentially continuous.

A property of continuous machines that can be propagated with relatively low effort and vastly simplifies implementation of operations such as composition is monotonicity in the following sense: Call a machine M **monotone** if $M(\varphi)(n, q') = \text{Some}\, a'$ implies that for any $m \geq n$ it holds that $M(\varphi)(m, q') = \text{Some}\, a'$. Call a continuous machine (M, μ) a **monotone machine** if M is monotone and μ **terminates with** M in the sense that once M returns a value, further increasing the effort on the same inputs does not lead μ to return bigger lists.

For a monotone machine M, the operator F_M is single-valued. The machine we used to implement inversion in Section 3.1 is not monotone. A continuous machine constructed from an oracle machine as outlined previously and those constructed from associates as outlined in the previous subsection are monotone. Thus, if equality on $\mathbf{Q}$ is decidable, translating from a continuous machine to an associate and back allows to force monotonicity. There is also a direct method that works without additional assumptions about question and answer sets.

▶ **Proposition 7.** *From a continuous machine (M, μ) a monotone machine that implements a choice function of F_M can be obtained. This construction can be done fully uniformly.*

Monotone machines are easier to operate on as it is not necessary to keep track of the exact value of an effort that leads to a return value but an upper bound is sufficient.

▶ **Theorem 8.** *From monotone machines (M, μ) and (M', μ') another monotone machine $(M' \circ_{\mu'} M, \mu \circ_M \mu')$ such that $F_{M' \circ_{\mu'} M} = F_{M'} \circ F_M$ can be obtained in a fully uniform way.*

Similar results hold for other basic operations. Continuous machines can be composed by first making them monotone and then composing them. There is also a more direct way of composing continuous machines but we failed to produce a simple description, the proofs of correctness are complicated and in experiments it did not perform well.

5 Conclusion

This paper is formulated from a point of view of computable analysis. Computable analysis traditionally investigates known theorems from analysis and functional analysis concerning their computational content. The mathematical background is developed over a classical meta-theory as are correctness proofs of algorithms. An important part of computable analysis is that incomputable and discontinuous functions are not excluded and the classification

of problems according to their degree of incomputability or discontinuity via Weihrauch reducibility is frequently studied [3]. Our work and the Incone library follow the traditions of computable analysis in the Coq development and as mathematicians we found working over a strong meta-theory convenient. A clear drawback is that providing computational content often means refining classical proofs and leads to some redundancy. However, starting with a classical proof and effectivize step by step is often instructive.

Represented spaces relate to concepts popular in constructive analysis: A representation defines a partial equivalence relation on its names by $\varphi \sim \psi \iff \delta(\varphi) = \delta(\psi)$. Conversely, given a partial equivalence relation on Baire space one can consider the quotient space and consider the quotient mapping a representation. Formulating everything using the equivalence relations, mentioning X can be avoided completely. This approach is for instance followed by developments like C-CoRn [6]. A function is called a morphism if it respects the equivalence relations and each such function induces a corresponding function on the equivalence classes that it realizes with respect to the quotient mapping as representation. Coq does not support quotient types and a direct description of the set of equivalence classes is additional information. Definability of a function on abstract description need no longer correspond to computability. Variations of this approach exist in Coq and other proof assistants under the name "refinements" [4, 19], but the objectives and with them which concepts are considered basic or useful differ significantly from our setting.

In our presentation we completely skipped the discussion of dialogue trees and jumped to associates directly. In work about total functionals and mathematical work, dialogue trees play a central role. Partial functions can be captured using a coinductive type of such trees. We decided against this due to negative experiences with coinduction in Coq, but we may try in the future. It may also be worth looking into sequentiality concerns closer: While continuous machines characterize a sequential model of computation, they are seemingly non-sequential as the computations for different efforts may take distinct paths and need not be increasing in any way.

References

1 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers–an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016.

2 Vasco Brattka and Guido Gherardi. Completion of choice. *arXiv preprint arXiv:1910.13186*, 2019.

3 Vasco Brattka, Guido Gherardi, and Arno Pauly. Weihrauch complexity in computable analysis. *arXiv preprint arXiv:1707.03202*, 2017.

4 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.

5 Robert L Constable. Type two computational complexity. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 108–121. ACM, 1973.

6 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

7 Damir D. Dzhafarov. Joins in the strong Weihrauch degrees. *Math. Res. Lett.*, 26(3):749–767, 2019. doi:10.4310/MRL.2019.v26.n3.a5.

8 Yannick Forster, Dominik Kirst, and Florian Steinberg. Towards Extraction of Continuity Moduli in Coq. *EUTYPES-TYPES 2020*, 2020.

9 Andrzej Grzegorczyk. On the definitions of computable real continuous functions. *Fund. Math.*, 44:61–71, 1957.

10 Martín Hötzel Escardó and Chuangjie Xu. The inconsistency of a brouwerian continuity principle with the curry–howard interpretation. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

11 Akitoshi Kawamura. *Computational complexity in analysis and geometry*. University of Toronto, 2011.

12 Stephen Cole Kleene. Countable functionals. *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam*, 1959.

13 Ker-I Ko. *Complexity theory of real functions*. Progress in Theoretical Computer Science. Birkhäuser Boston Inc., Boston, MA, 1991.

14 Michal Konecnỳ, Florian Steinberg, and Holger Thies. Continuous and monotone machines. *CoRR*, abs/2005.01624, 2020. `arXiv:2005.01624`.

15 Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in mathematics: Proceedings of the colloquium held at Amsterdam, 1957 (edited by A. Heyting)*, Studies in Logic and the Foundations of Mathematics, pages 101–128. North-Holland Publishing Co., Amsterdam, 1959.

16 Christoph Kreitz and Klaus Weihrauch. Theory of representations. *Theoretical computer science*, 38:35–53, 1985.

17 Daniel Lacombe. Sur les possibilités d'extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles. In *Le raisonnement en mathématiques et en sciences expérimentales*, Colloques Internationaux du Centre National de la Recherche Scientifique, LXX, pages 67–75. Editions du Centre National de la Recherche Scientifique, Paris, 1958.

18 Branimir Lambov. The basic feasible functionals in computable analysis. *Journal of Complexity*, 22(6):909–917, 2006.

19 Peter Lammich. Automatic data refinement. In *International Conference on Interactive Theorem Proving*, pages 84–99. Springer, 2013.

20 John Longley. The sequentially realizable functionals. *Annals of Pure and Applied Logic*, 117(1-3):1–93, 2002.

21 John Longley and Dag Normann. *Higher-order computability*, volume 100. Springer, 2015.

22 Dag Normann. Computability over the partial continuous functionals. *The Journal of Symbolic Logic*, 65(3):1133–1142, 2000.

23 Arno Pauly and Martin Ziegler. Relative computability and uniform continuity of relations. *J. Logic & Analysis*, 5, 2013.

24 Matthias Schröeder. *Admissible Representations for Continuous Computations*. PhD thesis, FernUniversität Hagen, 2002.

25 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. working paper or preprint, June 2019. URL: `https://hal.inria.fr/hal-02167423`.

26 Florian Steinberg, Laurent Thery, and Holger Thies. Quantitative continuity and computable analysis in coq. *arXiv preprint arXiv:1904.13203*, 2019.

27 Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1936. `doi:10.1112/plms/s2-42.1.230`.

28 Jaap Van Oosten et al. Partial combinatory algebras of functions. *Notre Dame Journal of formal logic*, 52(4):431–448, 2011.

29 Klaus Weihrauch. *Computable Analysis*. Springer, Berlin/Heidelberg, 2000.

30 Chuangjie Xu and Martín Escardó. A constructive model of uniform continuity. In *International Conference on Typed Lambda Calculi and Applications*, pages 236–249. Springer, 2013.

A Proofs

Proposition 3

Fix an enumeration of $\mathbf{Q}$. Let μ be the function that returns the minimal initial segment with respect to this enumeration such that the implication (1) is fulfilled. Since F is continuous, μ is well defined. It is a modulus by definition and it can be checked that it is also self-modulating.

Proposition 4

Let $d\colon \operatorname{seq}(\mathbf{Q}\times\mathbf{A})\to\operatorname{opt}(\mathcal{B})$ be a function that, if the input list is the graph of a finite function ϕ, returns some $\varphi\in\operatorname{dom}(F)$ such that $\phi=\varphi|_{\operatorname{dom}(\phi)}$ if such a φ exists and otherwise returns None. Let μ a self-modulating modulus of F that exists by Proposition 3 and $(q_n)_{n\in\mathbb{N}}$ an enumeration of $\mathbf{Q}$. Let $M(\varphi)(n,q')$ be given by

$$\begin{cases}\operatorname{Some}(F(\varphi')(q')) & \text{if } d(\varphi|_{(q_1,\dots,q_n)})=\operatorname{Some}\varphi'\\ & \text{and } \mu(\varphi')(q')\subseteq(q_1,\dots,q_n)\\ \operatorname{None} & \text{otherwise.}\end{cases}$$

If M returns something, the return value is correct because μ is self-modulating. On the other hand, whenever $\varphi\in\operatorname{dom}(F)$, there exists some n such that $\mu(\varphi)(q')\subseteq(q_1,\dots,q_n)$ and for this n the machine reproduces the value of F. Clearly, the values of M depend only on the values of φ on $(q_1,\dots,q_n)$, where n is such that $\mu(\varphi')(q')\subseteq(q_1,\dots,q_n)$. Just returning $(q_1,\dots,q_n)$ is a modulus of M that is independent of φ and thus self-modulating.

Proposition 7

Consider the monotone machine $\operatorname{uf}(M)$ (for "use first") defined as follows: on input of φ, n and q' search for the smallest $m\le n$ such that a return-value is produced and return this value, if no such m exists return None. As $\operatorname{uf}(M)$ is monotone, $F_{\operatorname{uf}(M)}$ is a partial function and it respects the interpretation of M in the sense that $F_{\operatorname{uf}(M)}$ is a choice function for the multivalued function F_M.

A version $\operatorname{uf}(\mu)$ of the modulus such that $(\operatorname{uf}(M),\operatorname{uf}(\mu))$ is a monotone machine can be defined by

$$\operatorname{uf}(\mu)(\varphi)(n,q') := \bigcup_{\{i\mid i\le n\wedge\forall j<i\colon\ M(\varphi)(j,q')=\operatorname{None}\}} \mu(\varphi)(i,q').$$

We omit the straight forward computation that this modulus is appropriate.

The modulus takes a union over all previous values, which leads to an undesirable overestimation. As a consequence, the modulus is monotone in the sense that the lists it returns grow with increasing effort and, while this can be a useful property, it is not required for the modulus of a monotone machine. One may be tempted to improve the construction by omitting the values of the modulus where M returns None. Unfortunately, the function obtained in this way is in general neither a modulus of $\operatorname{uf}(M)$ nor self-modulating.

Theorem 8

Let (M,μ) and (M',μ') be monotone machines such that $F_M\colon \mathbf{A}^{\mathbf{Q}}\rightrightarrows\mathbf{A}'^{\mathbf{Q}'}$ and $F_{M'}\colon \mathbf{A}'^{\mathbf{Q}'}\rightrightarrows\mathbf{A}''^{\mathbf{Q}''}$. Define the **monotone machine composition** as follows: First fix some default element $a'_d\in\mathbf{A}'$ and for each function $\varphi\colon\mathbf{Q}\to\mathbf{A}$ define a sequence of functions $\varphi'_n\colon\mathbf{Q}'\to\mathbf{A}'$ by

$$\varphi'_n(q') := \begin{cases}a' & \text{if } M(\varphi)(n,q')=\operatorname{Some}a'\\ a'_d & \text{otherwise.}\end{cases}$$

Use dom_n as shorthand for the set of elements $q' \in \mathbf{Q}'$ such that there exists an a' with $M(\varphi)(n, q') = \mathrm{Some}\, a'$. Note that whenever $\varphi \in \mathrm{dom}(F_M)$, then φ'_n and $F_M(\varphi)$ coincide on dom_n by these definitions. Let $(M' \circ_{\mu'} M)(\varphi)(n, q'')$ be

$$\begin{cases} M'(\varphi'_n)(n, q'') & \text{if } \mu'(\varphi'_n)(n, q'') \subseteq \mathrm{dom}_n \\ \mathrm{None} & \text{otherwise.} \end{cases}$$

Here, we put the index μ' at the composition as the outcome may be different for different valid moduli μ' of M'. Define the composition of the moduli by

$$(\mu \circ_M \mu')(\varphi)(n, q'') := \bigcup_{q' \in \mu'(\varphi'_n)(n, q'')} \mu(\varphi)(n, q').$$

Just like the composition of machines depends on μ', the composition of moduli depends on M via the definition φ'_n.

The above correctly implements composition. To see this, Let us first argue that the composition is monotone again. For this fix some inputs φ and q'' and assume that $(M' \circ_{\mu'} M)(\varphi)(n, q'') = \mathrm{Some}\, a''$. This can only be the case if $\mu'(\varphi'_n)(n, q'') \subseteq \mathrm{dom}_n$ and $M'(\varphi'_n)(n, q'') = \mathrm{Some}\, a''$. To prove monotonicity we need to show that the same is true if n is replaced by $n + 1$. Since M' is monotone it is sufficient to prove the list returned by the modulus to be included in dom_{n+1}. Since M is monotone, φ'_n and φ'_{n+1} coincide on dom_n. As μ' is a modulus of M', it holds that $M'(\varphi'_{n+1})(n, q'') = \mathrm{Some}\, a''$. Since μ' terminates with M', we get $\mu'(\varphi'_{n+1})(n + 1, q'') = \mu'(\varphi'_{n+1})(n, q'')$. Finally, using that μ is self-modulating, we conclude

$$\begin{aligned} \mu'(\varphi'_{n+1})(n + 1, q'') &= \mu'(\varphi'_{n+1})(n, q'') \\ &= \mu'(\varphi'_n)(n, q'') \subseteq \mathrm{dom}_n \subseteq \mathrm{dom}_{n+1}\,. \end{aligned}$$

We omit the details of how to verify that the modulus is appropriate, and only outline how to prove the more important half of the equality, namely that the left-hand of the equation extends the right-hand side. For this assume that the right-hand side is defined in φ. This means that $\varphi \in \mathrm{dom}(F_M)$ and $F_M(\varphi) \in \mathrm{dom}(F_{M'})$. Consider the sequence of functions φ'_n as defined above and note that since M is monotone and $\varphi \in \mathrm{dom}(F_M)$, this sequence converges to $F_M(\varphi)$. Since μ' is self-modulating, it is in particular sequentially continuous and therefore the sequence $\mu'(\varphi'_n)$ converges to $\mu'(F_M(\varphi))$. This means that for any fixed q'' we can first pick n big enough for $M'(F_M(\varphi))(n, q'')$ to take a value, then increase it further so that for all $k \geq n$ it holds that $\mu'(\varphi'_k)(n, q'') = \mu'(F_M(\varphi))(n, q'')$. As μ' terminates with M', further increasing n will no longer change the list it returns and we can use this to make sure that it is contained in dom_n as dom_n eventually contains every element of $\mathbf{Q}'$. As q'' was arbitrary, the left-hand side is defined and equal $F_{M'}(F_M(\varphi))$.