Robust computability notions for types arising in classical analysis

John Longley

School of Informatics
University of Edinburgh
jrl@inf.ed.ac.uk

British Logic Colloquium
Sussex, September 2017

Computability: three stages of enlightenment

Presented via crude slogans . . .

Computability: three stages of enlightenment

Presented via crude slogans . . .

1. All reasonable definitions of computability are equivalent.
   E.g. Turing machines, λ-calculus, Post processes, . . . all yield the same functions $\mathbb{N} \to \mathbb{N}$.

Computability: three stages of enlightenment

Presented via crude slogans ...

1. All reasonable definitions of computability are equivalent.
   E.g. Turing machines, λ-calculus, Post processes, ... all yield the same functions $\mathbb{N} \to \mathbb{N}$.
2. No they're not. If more complex kinds of 'data' are admitted (e.g. functions acting on functions acting on functions), then different flavours of computability are possible (Longley and Normann 2015). Programming languages differ in expressivity!

Computability: three stages of enlightenment

Presented via crude slogans ...

1. All reasonable definitions of computability are equivalent.
   E.g. Turing machines, λ-calculus, Post processes, ... all yield the same functions $\mathbb{N} \to \mathbb{N}$.
2. No they're not. If more complex kinds of 'data' are admitted (e.g. functions acting on functions acting on functions), then different flavours of computability are possible (Longley and Normann 2015). Programming languages differ in expressivity!
3. Yes they are! Under quite mild conditions, different flavours of higher-order computability give rise to exactly the same hereditarily total functionals over $\mathbb{N}$, for non-trivial reasons (Normann 2000, Longley 2007).

Computability: three stages of enlightenment

Presented via crude slogans ...

1. All reasonable definitions of computability are equivalent.
   E.g. Turing machines, λ-calculus, Post processes, ... all yield the same functions $\mathbb{N} \to \mathbb{N}$.
2. No they're not. If more complex kinds of 'data' are admitted (e.g. functions acting on functions acting on functions), then different flavours of computability are possible (Longley and Normann 2015). Programming languages differ in expressivity!
3. Yes they are! Under quite mild conditions, different flavours of higher-order computability give rise to exactly the same hereditarily total functionals over $\mathbb{N}$, for non-trivial reasons (Normann 2000, Longley 2007).

This talk: Some recent extensions of the scope of phenomenon 3, covering a range of data types relevant to 'mathematical practice', especially in analysis.

Foundational issues

To define 'computability' over e.g. spaces of analytic functions, we need to face the foundational question: What sort of entity *is* an analytic function anyway?

Foundational issues

To define ‘computability’ over e.g. spaces of analytic functions, we need to face the foundational question: What sort of entity *is* an analytic function anyway?

- A certain kind of set, as in ZF? Maybe, but hasn’t proved very fruitful for developing theories of (effective) computability.

Foundational issues

To define 'computability' over e.g. spaces of analytic functions, we need to face the foundational question: What sort of entity *is* an analytic function anyway?

- A certain kind of set, as in ZF? Maybe, but hasn't proved very fruitful for developing theories of (effective) computability.
- An object of finite type, as in Church's simple theory of types? Much better for our purposes; has also proved convenient for formalizing mathematics as in Isabelle/HOL.

Foundational issues

To define 'computability' over e.g. spaces of analytic functions, we need to face the foundational question: What sort of entity *is* an analytic function anyway?

- A certain kind of set, as in ZF? Maybe, but hasn't proved very fruitful for developing theories of (effective) computability.
- An object of finite type, as in Church's simple theory of types? Much better for our purposes; has also proved convenient for formalizing mathematics as in Isabelle/HOL.

Idea: Types are built up e.g. from a base type $\mathbb{N}$ via a 'function space' constructor $\rightarrow$ (admitting various interpretations). So e.g.

- Natural numbers / rationals are representable at type $\mathbb{N}$.
- Real / complex numbers are representable at type $\mathbb{N} \rightarrow \mathbb{N}$.
- Functions on $\mathbb{R}$ or $\mathbb{C}$ are representable at $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$.
- Operators on such functions are representable at $\ldots$, etc.

'Feferman's thesis': Most of analysis needs just the first few levels.

Subset and quotient types

For ‘practical’ purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Subset and quotient types

For ‘practical’ purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Not so in constructive or computable settings. E.g. under any reasonable definition of ‘computability’ . . .

Subset and quotient types

For 'practical' purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Not so in constructive or computable settings. E.g. under any reasonable definition of 'computability' ...

- $f \mapsto \min i.\ f(i) \neq 0$ is computable on $(\mathbb{N} \to \mathbb{N}) - \{\Lambda i.0\}$, but not extendable to a computable (or continuous) function on $\mathbb{N} \to \mathbb{N}$.

Subset and quotient types

For 'practical' purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Not so in constructive or computable settings. E.g. under any reasonable definition of 'computability' . . .

- $f \mapsto \min i.\ f(i) \neq 0$ is computable on $(\mathbb{N} \to \mathbb{N}) - \{\Lambda i.0\}$, but not extendable to a computable (or continuous) function on $\mathbb{N} \to \mathbb{N}$.
- $x \mapsto 1/x : \mathbb{R} - \{0\} \to \mathbb{R}$ is computable, but not extendable to a computable (or continuous) function $\mathbb{R} \to \mathbb{R}$.

Subset and quotient types

For 'practical' purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Not so in constructive or computable settings. E.g. under any reasonable definition of 'computability' ...

- $f \mapsto \min i.\ f(i) \neq 0$ is computable on $(\mathbb{N} \to \mathbb{N}) - \{\Lambda i.0\}$, but not extendable to a computable (or continuous) function on $\mathbb{N} \to \mathbb{N}$.
- $x \mapsto 1/x : \mathbb{R} - \{0\} \to \mathbb{R}$ is computable, but not extendable to a computable (or continuous) function $\mathbb{R} \to \mathbb{R}$.
- Given a closed curve c in the plane and a point p not on c, can compute the winding number of c around p. Not extendable to a computable operation on arbitrary pairs (c, p).

Subset and quotient types

For ‘practical’ purposes, helpful to augment our system with subset and quotient types. E.g. $\mathbb{R}$ as a quotient of a subset of $\mathbb{N} \to \mathbb{N}$.

In the context of a classical logic (as in Isabelle/HOL), this is an inessential extension: e.g. a function with domain $S \subseteq \mathbb{N} \to \mathbb{N}$ can always be represented by some function on $\mathbb{N} \to \mathbb{N}$.

Not so in constructive or computable settings. E.g. under any reasonable definition of ‘computability’ . . .

- $f \mapsto \min i.\ f(i) \neq 0$ is computable on $(\mathbb{N} \to \mathbb{N}) - \{\Lambda i.0\}$, but not extendable to a computable (or continuous) function on $\mathbb{N} \to \mathbb{N}$.
- $x \mapsto 1/x : \mathbb{R} - \{0\} \to \mathbb{R}$ is computable, but not extendable to a computable (or continuous) function $\mathbb{R} \to \mathbb{R}$.
- Given a closed curve c in the plane and a point p not on c, can compute the winding number of c around p. Not extendable to a computable operation on arbitrary pairs (c, p).
- If f is analytic on a disc $D_{1+\epsilon}$ and nonzero on ∂D_1, can compute the number of zeros (by multiplicity) of f within D_1. Not extendable to arbitrary continuous f, if codomain is taken to be $\mathbb{N}$ rather than $\mathbb{R}$.

Robust computability notions for mathematical types

Moral: Saying what 'computability' means at type $S \to T$ doesn't immediately fix what it means at $S' \to T$ where $S \subseteq T$.

So a 'computability theory' applicable e.g. to analysis should pay due attention to subset types (perhaps overlooked so far). Quotient types then fall out for general abstract reasons.

Robust computability notions for mathematical types

Moral: Saying what 'computability' means at type $S \to T$ doesn't immediately fix what it means at $S' \to T$ where $S \subseteq T$.

So a 'computability theory' applicable e.g. to analysis should pay due attention to subset types (perhaps overlooked so far). Quotient types then fall out for general abstract reasons.

Earlier work (Normann, Longley): Under mild conditions, two 'higher-order computability models' (e.g. programming languages) yield same total functions at all simple types (built from $\mathbb{N}$ via $\to$).

Present work: This remains largely true even when subset formation is thrown in. (Precise extent still being clarified, but covers naturally arising mathematical types.)

Robust computability notions for mathematical types

Moral: Saying what 'computability' means at type $S \to T$ doesn't immediately fix what it means at $S' \to T$ where $S \subseteq T$.

So a 'computability theory' applicable e.g. to analysis should pay due attention to subset types (perhaps overlooked so far). Quotient types then fall out for general abstract reasons.

Earlier work (Normann, Longley): Under mild conditions, two 'higher-order computability models' (e.g. programming languages) yield same total functions at all simple types (built from $\mathbb{N}$ via $\to$).

Present work: This remains largely true even when subset formation is thrown in. (Precise extent still being clarified, but covers naturally arising mathematical types.)

Other work: Much existing work on computability in analysis (e.g. Weihrauch) tends to pick some particular underlying 'model of computation' and see what that yields.
Our contribution is to show that the classes of 'computable functions' we get are (largely) independent of the choice of underlying model.

Interlude: Why is this interesting?

Some possible reasons:

Interlude: Why is this interesting?

Some possible reasons:

1. Sheds light on (theoretical) expressive power of different programming languages within the area of exact computation (here we represent reals via potentially infinite digit streams).

Interlude: Why is this interesting?

Some possible reasons:

1. Sheds light on (theoretical) expressive power of different programming languages within the area of exact computation (here we represent reals via potentially infinite digit streams).
2. Broadly relevant to questions of mathematical ontology: e.g. relates different 'constructive presentations' of mathematical objects.

Interlude: Why is this interesting?

Some possible reasons:

1. Sheds light on (theoretical) expressive power of different programming languages within the area of exact computation (here we represent reals via potentially infinite digit streams).
2. Broadly relevant to questions of mathematical ontology: e.g. relates different 'constructive presentations' of mathematical objects.
3. Relevant to: How 'computable' or 'mechanistic' is your favourite model of physics? Cf. Laplace's demon.

Some technical details: Higher-order computation models

Types: $\sigma ::= \mathtt{N} \mid \sigma \to \sigma$.

Our computation models are typed partial combinatory algebras with weak numerals and ground-type iteration. They consist of:

- a set $A(\sigma)$ for each type σ,
- for each σ, τ, a partial function $\cdot_{\sigma\tau} : A(\sigma \to \tau) \times A(\sigma) \rightharpoonup A(\tau)$ (called 'application'),
- elements $k_{\sigma\tau} \in A(\sigma \to \tau \to \sigma)$, and $s_{\rho\sigma\tau} \in A(\cdots)$,
- elements $\widehat{0}, \widehat{1}, \widehat{2}, \ldots \in A(\mathtt{N})$, $suc \in A(\mathtt{N} \to \mathtt{N})$ and $primrec \in A(\mathtt{N} \to (\mathtt{N} \to \mathtt{N} \to \mathtt{N}) \to \mathtt{N} \to \mathtt{N})$,
- an element $iter \in A((\mathtt{N} \to \mathtt{N}) \to (\mathtt{N} \to \mathtt{N}))$

. . . all satisfying various axioms.

There is an abundance of such structures, both 'syntactic' (term models for programming languages) and 'semantic' (arising from domain theory, game semantics, . . .), embodying different flavours of higher-order computability.

Some of our results also work in a non-deterministic variant of the above setup (new progress).

Representing 'mathematical' objects within A

An *A-assembly* X consists of:

- a set $|X|$,
- a type σ_X,
- a *realizability relation* $\Vdash_X \subseteq A(\sigma_X) \times |X|$, such that $\forall x. \exists a. \ a \Vdash_X x$. (Think of x and a as 'mathematical' and 'computational' objects respectively.)

Our intended operations for constructing 'mathematical' types can be interpreted in terms of A-assemblies:

- Start with $N = (\mathbb{N}, \mathtt{N}, \Vdash_N)$, where $a \Vdash_N n$ iff $a = \widehat{n}$.
- Given assemblies X, Y, may form an assembly $X \Rightarrow Y$ whose elements are functions $f : |X| \to |Y|$ that are 'realized' by some $t \in A(\sigma_X \to \sigma_Y)$ in an evident sense.
- Given an assembly X and a subset $S \subset |X|$, may form the restricted assembly $\mathrm{Sub}(X, S)$ in an obvious way.

Idea is to see how this interpretation of our types looks for different computation models A.

Axioms on computation models

We obtain results of interest under various combinations of axioms on A. (Cleaner than approach via simulations in Longley 2007.)

E.g. the following axioms can be seen as capturing typical intrinsic consequences (for A) of effective simulability.
Here Δ_A denotes $|N \Rightarrow N|$ (this plays a key role).

Enumeration: For all $f \in A(\mathtt{N} \to \mathtt{N})$ there exists $g \in \Delta_A$ such that for all $n, m \in \mathbb{N}$ we have: $f \cdot \widehat{n} = \widehat{m}$ iff $\exists i.\, g(i) = \langle n, m \rangle + 1$.

(N.B. Only rarely holds 'uniformly within A'.)

Collection (w.r.t. a type σ): For any $\Phi \in A(\sigma \to \mathtt{N})$, there exists $f \in A(\mathtt{N} \to \mathtt{N})$ with the same 'range in $\mathbb{N}$' as Φ. More precisely, for any $m \in \mathbb{N}$, we have $\exists a \in A(\sigma).\ \Phi \cdot a = \widehat{m}$ iff $\exists n \in \mathbb{N}.\ f \cdot \widehat{n} = \widehat{m}$.

(Again, rarely holds uniformly.)

More 'standard' axioms: Continuity, Normalizability, Restriction.

Regular types

A function F even of high type can typically be represented by a function $g_F : \mathbb{N} \to \mathbb{N}$, called a graph for F (well understood).

We call a mathematical type T regular (for A) if its interpretation over A contains exactly those functions that have a graph in Δ_A. So if T is regular, its contents are completely determined by Δ_A.

- Second-order math types: e.g. take $Q \subseteq N$, form $Q \Rightarrow N$, take $R \subseteq Q \Rightarrow N$ and form $R \Rightarrow N$. Various types of this form are regular under various combinations of axioms, via abstract versions of the Kreisel-Lacombe-Shoenfield argument.
- Third-order and above: requires the heavy Normann-Longley machinery. Details still being worked out, but under reasonable axioms, certainly get regularity when all subsets involved are tame (i.e. Δ-separable).

  [N.B. Here we need a bit more computing power in A: type 1 recursion rather than just ground type iteration.]

So if A and B satisfy certain axioms and $\Delta_A = \Delta_B$, they'll agree on all math types that can be proved regular under these axioms.

Concluding remarks

- Questions of robustness/canonicity of computability concepts are characteristic of computability theory, whatever entities we're wanting to compute with.
- We're making some progress in establishing the existence of robust computability concepts for many types arising in analysis. Also have some counterexamples to indicate the limits of this phenomenon.
- The entities we compute with must themselves have at least a 'semi-constructive' character. E.g. can compute with arbitrary continuous functions $\mathbb{R} \to \mathbb{R}$, but not discontinuous ones.
- Our computability concepts may still not be the *only* reasonable ones for the types in question (cf. M. Schröder). More work needed here.

Draft paper available:

`homepages.inf.ed.ac.uk/jrl/Research/ubiquity-reloaded3.pdf`