Theory of Real Computation according to EGC*

Chee Yap
Courant Institute of Mathematical Sciences
Department of Computer Science
New York University

April 17, 2007

Abstract

The Exact Geometric Computation (EGC) mode of computation has been developed over the last decade in response to the widespread problem of numerical non-robustness in geometric algorithms. Its technology has been encoded in libraries such as LEDA, CGAL and Core Library. The key feature of EGC is the necessity to decide zero in its computation. This paper addresses the problem of providing a foundation for the EGC mode of computation. This requires a theory of real computation that properly addresses the Zero Problem. The two current approaches to real computation are represented by the analytic school and algebraic school. We propose a variant of the analytic approach based on **real approximation**.

- To capture the issues of representation, we begin with a reworking of van der Waerden's idea of explicit rings and fields. We introduce explicit sets and explicit algebraic structures.
- Explicit rings serve as the foundation for real approximation: our starting point here is not $\mathbb{R}$, but $\mathbb{F} \subseteq \mathbb{R}$, an explicit ordered ring extension of $\mathbb{Z}$ that is dense in $\mathbb{R}$. We develop the approximability of real functions within standard Turing machine computability, and show its connection to the analytic approach.
- Current discussions of real computation fail to address issues at the intersection of continuous and discrete computation. An appropriate computational model for this purpose is obtained by extending Schönhage's pointer machines to support both algebraic and numerical computation.
- Finally, we propose a synthesis wherein both the algebraic and the analytic models coexist to play complementary roles. Many fundamental questions can now be posed in this setting, including transfer theorems connecting algebraic computability with approximability.

1 Introduction

Software breaks down due to numerical errors. We know that such breakdown has a numerical origin because when you tweak the input numbers, the problem goes away. Such breakdown may take the dramatic form of crashing or looping. But more insidiously, it may silently produce *qualitatively* wrong results. Such qualitative errors are costly to catch further down the data stream. The economic consequences of general software errors have been documented[1] in a US government study [32]. Such problems of geometric nonrobustness are also well-known to practitioners, and to users of geometric software. See [22] for the anatomy of such breakdowns in simple geometric algorithms.

In the last decade, an approach called **Exact Geometric Computation** (EGC) has been shown to be highly effective in eliminating nonrobustness in a large class of basic geometric problems. The fundamental analysis and prescription of EGC may be succinctly stated as follows:

*Expansion of a talk by the same title at Dagstuhl Seminar on "Reliable Implementation of Real Number Algorithms: Theory and Practice", Jan 7-11, 2006. This work is supported by NSF Grant No. 043086.

[1] A large part of the report focused on the aerospace and automobile industries. Both industries are major users of geometric software such as CAD modelers and simulation systems. The numerical errors in such software are well-known and so we infer that part of the cost comes from the kind of error of interest to us.

"Geometry is concerned with relations among geometric objects. Basic geometric objects (e.g., points, half-spaces) are parametrized by numbers. Geometric algorithms (a) construct geometric objects and (b) determine geometric relations. In real geometry, these relations are determined by evaluating the signs of real functions, typically polynomials. Algorithms use these signs to branch into different computation paths. Each path corresponds to a particular output geometry. So the EGC prescription *says that, in order to compute the correct geometry, it is sufficient to ensure that the correct path is taken. This reduces to error-free sign computations in algorithms."*

How do algorithms determine the sign of a real quantity x? Typically, we compute approximations $\widetilde{x}$ with increasing precision until the error $|x-\widetilde{x}|$ is known to be less than $|\widetilde{x}|$; then we conclude $\mathtt{sign}(x) = \mathtt{sign}(\widetilde{x})$. Note that this requires interval arithmetic, to bound the error $|x-\widetilde{x}|$. But in case $x = 0$, interval arithmetic does not help – we may reduce the error as much as we like, but the stopping condition (i.e., $|x-\widetilde{x}| < |\widetilde{x}|$) will never show up. This goes to the heart of EGC computation: how to decide if $x = 0$ [42]. This **zero problem** has been extensively studied by Richardson [33, 34]. Numerical computation in this **EGC mode**[2] has far reaching implications for software and algorithms. Currently software such as LEDA [11, 27], CGAL [17] and the Core Library [21] supports this EGC mode. We note that EGC assumes that the numerical input is exact (see discussion in [42]).

In this paper, we address the problem of providing a computability theory for the EGC mode of computation. Clearly, EGC requires arbitrary precision computation and falls under the scope of the theory of real computation. While the theory of computation for discrete domains (natural numbers $\mathbb{N}$ or strings Σ^*) has a widely accepted foundation, and possesses a highly developed complexity theory, the same cannot be said for computation over an uncountable and continuous domain such as $\mathbb{R}$. Currently, there are two distinct approaches to the theory of real computation. We will call them the **analytic school** and the **algebraic school** respectively.

The analytic school goes back to Turing (1936), Grzegorczyk (1955) and Lacombe (1955) (see [39]). Modern proponents of this theory include Weihrauch [39], Ko [23], Pour-El and Richards [31] and others. In fact, there are at least six equivalent versions of analytic computability, depending on one's preferred starting point (metric spaces, domain-theory, etc) [20, p. 330]. In addition, there are complementary logical or descriptive approaches, based on algebraic specifications or equations. (e.g., [20]). But approaches based on Turing machines are most congenial to our interest in complexity theory. Here, real numbers are represented by rapidly converging Cauchy sequences. But an essential extension of Turing machines is needed to handle such inputs. In Weihrauch's TTE approach [39], Turing machines are allowed to compute forever to handle infinite input/output sequences. For the purposes of defining complexity, we prefer Ko's variant [23], using oracle Turing machines that compute in finite time. There is an important branch of the analytic school, sometimes known[3] as the Russian Approach [39, Chap. 9]. Below, we will note the close connections between the Russian Approach (Kolmogorov, Uspenskiĭ, Mal'cev) and our work.

The algebraic school goes back to the study of algebraic complexity [7, 10]. The original computational model here is non-uniform (e.g., straightline programs). The uniform version of this theory has been advocated as a theory of real computation by Blum, Shub and Smale [6]. Note that this uniform algebraic model is also the *de facto* computational model of theoretical computer science and algorithms. Here, the preferred model is the Real RAM [2], which is clearly equivalent to the BSS model. In the algebraic school, real numbers are directly represented as atomic objects. The model allows the primitive operations to be carried out in a single step, without error. We can also compare real numbers without error. Although the BSS computational model [6] emphasizes ring operations $(+, -, \times)$, in the following, we will allow our algebraic model to use any chosen set Ω of real operations.

We have noted that the zero problem is central to EGC: sign determination implies zero determination. Conversely, if we can determine zero, we can also determine sign under mild assumptions (though the complexity can be very different). A natural hierarchy of zero problems can be posed [42]. This hierarchy can be used to give a natural complexity classification of geometric problems. But these distinctions are lost in the analytic and algebraic approaches: the zero problem is undecidable in analytic approach ([23, Theorem 2.5, p. 44]); it is trivial in the algebraic computational model. We need a theory where the complexity of

[2]Likewise, one may speak of the "numerical analysis mode", the "computer algebra mode", or the "interval arithmetic mode" of computing. See [41].

[3]We might say the main branch practices the Polish Approach.

the zero problems is more subtly portrayed, consistent with practical experience in EGC computation.

In numerical analysis, it is almost axiomatic that input numbers are inexact. In particular, this justifies the **backward analysis** formulation of solving problems. On the other hand, the EGC approach and the zero problem makes sense only if numerical inputs are exact. Hence, we must reject the oft-suggested notion that *all real inputs are inherently inexact*. In support of this notion, it is correctly noted that all physical constants are inherently inexact. Nevertheless, there are many applications of numerical computing for which such a view is untenable, in areas such as computer algebra, computational number theory, geometric theorem proving, computational geometry, and geometric modeling. The exact input assumption is also the dominant view in the field of algorithmics [13]. We also refute a related suggestion, that *the backward analysis solution is inevitable for real input problems*. A common argument to support this view says that "even if the input is exact, computers can only compute real quantities with limited precision". But the success of EGC, and existence of software such as LEDA, CGAL and Core Library, provides a wealth of counter examples. Nevertheless, the backward analysis problem formulation has an important advantage over EGC: it does not require a model of computation that must be able to decide zero. But the point of this paper is to shed some light on those computational problems for which we must decide zero.

The goal of this paper is three-fold: First, we propose a variant of the analytic approach that is suitable for studying the zero problem and for modeling contemporary practice in computation. Second, we propose a computational model suitable for "semi-numeric problems" – these are problems such as in computational geometry or topology where the input and output comprise a combination of numeric and discrete data. This is necessary to formalize what we mean by computation in the EGC mode. Finally, we propose a framework whereby the algebraic approach can be discussed in its natural relation to the analytic approach. Basic questions such as "transfer theorems" can be asked here.

2 Explicit Set Theory

Any theory of computation ought to address the representation of the underlying mathematical domains, especially with a view to their computation on machines. Thus Turing's 1936 analysis of the concept of computability began with a discussion of representation of real numbers. But the issue of representation usually is de-emphasized in discrete computability theory because we usually define our problems directly[4] on a canonical universe such as strings (Σ^* in Turing computability) or natural numbers ($\mathbb{N}$ in recursive function theory). Other domains (e.g., finite graphs) must be suitably encoded in this canonical universe. The encoding of such discrete domains does not offer conceptual difficulties, although it may have significant complexity theoretic implications. But for continuous or uncountable domains such as $\mathbb{R}$, choice of representation may be critical (e.g., representing reals by its binary expansion is limiting [39]). See Weihrauch and Kreitz [24, 39] for topological investigations of representations (or "naming systems") for such sets.

The concept of representation is implicit (sic) in van der Waerden's notion of "explicit rings and fields" (see Fröhlich and Shepherdson [18]). To bring representation into algebraic structures, we must first begin with the more basic concept of "explicit sets". These ideas were extensively developed by the Russian school of computability. Especially relevant is Mal'cev's theory of **numbered algebraic systems** [26, Chapters 18,25,27]. A **numbering** of an arbitrary set S is an onto, total function $\alpha : D \to S$ where $D \subseteq \mathbb{N}$. The pair (S, α) is called a **numbered set**. A numbered algebraic system, then, is an algebraic system (see below) whose carrier set S has a numbering α, and whose functions and predicates are represented by functions on $\mathbb{N}$ that are compatible with α in the natural way.

Intuitively, an explicit set is one in which we can recognize the identity of its elements through representation elements. This need for recognizing elements of a set is well-motivated by EGC's need to decide zero "explicitly", and to perform exact operations. Explicitness is a form of effectivity. Traditionally, one investigates effectivity of subsets of $\mathbb{N}$, and such sets can be studied via Kleene or Gödel numberings. But the issues are more subtle when we treat arbitrary sets. Unlike $\mathbb{N}$, which has a canonical representation, general sets which arise in algebra or analysis may be specified in a highly abstract (prescriptive) manner that gives no hint as to their representation. A central motivation of Mal'cev [26, p. 287] is to study numberings of arbitrary sets. His theorems are inevitably about numbered sets, not just the underlying sets (cf. [26,

[4] I.e., we simply say that our problems are functions or relations on this canonical universe, thereby skipping the translation from the canonical universe to our intended domain of application.

Chapter 25]). Our point of departure is the desire to have a numbering-independent notion of explicitness. We expect that in a suitable formulation of explicit sets, certain basic axioms in naive set theory [19] will become theorems in explicit set theory. E.g., the well-ordering principle for sets.

Partial functions. If S, T are sets, we shall denote[5] a partial function from T to S by $f : T \dashrightarrow S$ (dashed arrow). If $f(x)$ is undefined, we write $f(x) = \uparrow$; otherwise we write $f(x) = \downarrow$. Call T the **nominal domain** of f; the **proper domain** of f is $\texttt{domain}(f) := \{x \in T : f(x) = \downarrow\}$. Relative to f, we may refer to $x \in T$ as **proper** if $f(x) = \downarrow$; otherwise x is **improper**. If $\texttt{domain}(f) = T$, then f is **total**, and we indicate total functions in the usual way, $f : T \to S$. Similarly, the sets S and $\texttt{range}(f) := \{f(s) : s \in \texttt{domain}(f)\}$ are (resp.) the **nominal** and **proper ranges** of f. We say f is **onto** if $\texttt{range}(f) = S$, and f is **1-1** if $f(x) = f(y) \neq \uparrow$ implies $x = y$.

In composing partial functions, we use the standard rule that a function value is undefined if any input argument is undefined. We often encounter predicates such as "$f(x) = g(y)$". We interpret such equalities in the "strong sense", meaning that the left side $f(x)$ is defined iff the right side $g(y)$ is defined. To indicate this **strong equality**, we write "$f(x) \equiv g(y)$".

Partial recursive partial functions. Partial functions on strings, $f : \Sigma^* \dashrightarrow \Sigma^*$ have a special status in computability theory: we say f is **partial recursive** if there is a (deterministic) Turing machine that, on input $w \in \Sigma^*$, halts with the string $f(w)$ in the output tape if $f(w) = \downarrow$, and does not halt (i.e., loops) if $f(w) = \uparrow$. If f is a total function, and f is partial recursive, then we say that f is **total recursive**. Both these definitions are standard. We need an intermediate version of these two concepts: assume that our Turing machines have two special states, $q_\downarrow$ (**proper state**) and $q_\uparrow$ (**improper state**). A Turing machine is **halting** if it halts on all input strings. We say[6] that f is **recursive** if there is a halting Turing machine M that, for all $w \in \Sigma^*$, if $f(w) = \uparrow$, M halts in the improper state $q_\uparrow$; if $f(w) = \downarrow$, then M halts in the proper state $q_\downarrow$ with the output tape containing $f(w)$. If *TOT-REC* (resp., *PART-REC*, *REC*) denote the set of total recursive (resp., partial recursive, recursive) functions, then we have $\textit{TOT-REC} \subseteq \textit{REC} \subseteq \textit{PART-REC}$. These inclusions are all strict.

Let $S \subseteq \Sigma^*$. For our purposes, we define the **characteristic function** of S to be $\chi_S : \Sigma^* \dashrightarrow \{1\}$ where $\chi_S(w) = 1$ if $w \in S$, and $\chi_S(w) = \uparrow$ otherwise. A closely related function is the **partial identity function**, $\iota_S : \Sigma^* \dashrightarrow S$ where $\iota_S(w) = w$ iff $w \in S$ and $\iota_S(w) = \uparrow$ otherwise. We say[7] S is (partial) **recursive** if χ_S (equivalently ι_S) is (partial) recursive.

Notes on Terminology. In conventional computability, it would be redundant to say "partial recursive *partial* function". Likewise, it would be an oxymoron to call a partial function "recursive". Note that a recursive function f in our sense is equivalent to "f is partial recursive with a recursive domain" in standard terminology. Our notion of recursive function anticipates its use in various concepts of "explicitness". In our applications, we are not directly computing over Σ^*, but are computing over some abstract domain S that is represented through Σ^*. Thus, our Turing machine computation over Σ^* is interpreted via this representation. We use the explicitness terminology in association with such concepts of "interpreted computations" – explicitness is a form of "interpreted effectivity". In recursive function theory, we know that the treatment of partial functions is an essential feature. In algebraic computations, we have a different but equally important reason for treating of partial functions: algebraic operations (e.g., division, squareroot, logarithm) are generally partial functions. The standard definitions would classify division as "partial recursive" (more accurately, partially explicit). But all common understanding of recursiveness dictates that division be regarded as "recursive" (more accurately, explicit). Thus, we see that our terminology is more natural.

In standard computability theory (e.g., [35]), the terms "computable" and "recursive" are often interchangeable. In this paper, the terms "recursive" and "partial recursive" are only applied to computing over

[5]This notation is used, e.g., by Weihrauch [24] and Mueller [29]. We thank Norbert Mueller for his contribution of this partial function symbol. An alternative notation is $f :\subseteq T \to S$ [39].

[6]In the literature, "total recursive" is normally shortened to "recursive"; so our definition of "recursive" forbids such an abbreviation.

[7]It is more common to say that S is recursively enumerable if χ_S is partial recursive. But note that Mal'cev [26, p.164–5] also calls S partial recursive when χ_S is partial recursive.

strings (Σ^*), "explicit" refer to computing over some abstract domain (S). Further, we reserve the term "computability" for real numbers and real functions (see Section 4), in the sense used by the analytic school [39, 23].

Representation of sets. We now consider arbitrary sets S, T. Call $\rho : T \dashrightarrow S$ a **representation** of S (with **representing set** T) if ρ is an onto function. If $\rho(t) = s$, we call t a **representation element** of s. Relative to ρ, we say t, t' are **equivalent**, denoted $t \equiv t'$ if $\rho(t) \equiv \rho(t')$ (recall $\equiv$ means equality is in the strong sense). In case $T = \Sigma^*$ for some alphabet Σ, we call ρ a **notation**. It is often convenient to identify Σ^* with $\mathbb{N}$, under some bijection that takes $n \in \mathbb{N}$ to $\underline{n} \in \Sigma^*$. Hence a representation of the form $\rho : \mathbb{N} \dashrightarrow S$ is also called a notation.

We generally use 'ν' instead of 'ρ' for notations. For computational purposes (according to Turing), we need notations. Our concept of notation ν and Mal'cev's numbering α are closely related: the fact that $\mathtt{domain}(\alpha) \subseteq \mathbb{N}$ while $\mathtt{domain}(\nu) \subseteq \Sigma^*$ is not consequential since we may identify $\mathbb{N}$ with Σ^*. But it is significant that ν is partial, while α is total. Unless otherwise noted, we may (wlog) assume $\Sigma = \{0, 1\}$. Note that if a set has a notation then it is countable.

A notation ν is **recursive** if the set[8] $E_\nu := \{(w, w') \in (\Sigma^*)^2 : \nu(w) \equiv \nu(w')\}$ is recursive. In this case, we say S is **ν-recursive**. If S is ν-recursive then the set $D_\nu := \{w \in \Sigma^* : \nu(w) = \downarrow\}$ $(= \mathtt{domain}(\nu))$ is recursive: to see this, note that $w \in D_\nu$ iff $(w, w_\uparrow) \notin E_\nu$ where $w_\uparrow$ is any word such that $\nu(w_\uparrow) = \uparrow$.

It is important to note that "recursiveness of ν" does not say that the function ν is a recursive function. Indeed, such a definition would not make sense unless S is a set of strings. The difficulty of defining explicit sets amounts to providing a substitute for defining "recursiveness of ν" as a function. Tentatively, suppose we say S is "explicit" (in quotes) if there "exists" a recursive notation ν for S. Clearly, the "existence" here cannot have the standard understanding – otherwise we have the trivial consequence that a set S is "explicit" iff it is countable. One possibility is to understand it in some intuitionistic sense of "explicit existence" (e.g., [4, p. 5]). But we prefer to proceed classically.

To illustrate some issues, consider the case where $S \subseteq \mathbb{N}$. There are two natural notations for S: the **canonical notation** of S is $\nu_S : \mathbb{N} \dashrightarrow S$ where $\nu_S(n) = n$ if $n \in S$, and otherwise $\nu_S(n) = \uparrow$. The **ordered notation** of S is $\nu'_S : \mathbb{N} \dashrightarrow S$ where $\nu'_S(n) = i$ if $i \in S$ and the set $\{j \in S : j < i\}$ has i elements. Let S be the halting set $K \subseteq \mathbb{N}$ where $n \in K$ iff the nth Turing machine on input string $\underline{n}$ halts. The canonical notation ν_K is not "explicit" since E_{ν_K} is not recursive. But the ordered notation ν'_K is "explicit" since $E_{\nu'_K}$ is the trivial diagonal set $\{(n, n) : n \in \mathbb{N}\}$. On the other hand, ν'_K does not seem to be a "legitimate" way of specifying notations (for instance, it is even not a computable function). The problem we face is to distinguish between notations such as ν_K and ν'_K.

Our first task is to distinguish a legitimate set of notations. We consider three natural ways to construct notations: let $\nu_i : \Sigma_i^* \dashrightarrow S_i$ $(i = 1, 2)$ be notations and $\#$ is a symbol not in $\Sigma_1 \cup \Sigma_2$.

1. (Cartesian product) The notation $\nu_1 \times \nu_2$ for the set $S_1 \times S_2$ is given by:

$$\nu_1 \times \nu_2 : (\Sigma_1^* \cup \Sigma_2^* \cup \{\#\})^* \dashrightarrow S_1 \times S_2$$

   where $(\nu_1 \times \nu_2)(w_1 \# w_2) = (\nu_1(w_1), \nu_2(w_2))$ provided $\nu_i(w_i) = \downarrow$ $(i = 1, 2)$; for all other w, we have $(\nu_1 \times \nu_2)(w) = \uparrow$.

2. (Kleene star) The notation ν_1^* for finite strings over S_1 is given by:

$$\nu_1^* : (\Sigma_1^* \cup \{\#\})^* \dashrightarrow S_1^*$$

   where $\nu_1^*(w_1 \# w_2 \# \cdots \# w_n) = \nu_1(w_1)\nu_1(w_2) \cdots \nu_1(w_n)$ provided $\nu_1(w_j) = \downarrow$ for all j; for all other w, we have $(\nu_1^*)(w) = \uparrow$.

3. (Restriction) For an arbitrary function $f : \Sigma_1^* \dashrightarrow \Sigma_2^*$, we obtain the following notation

$$\nu_2|_f : \Sigma_1^* \dashrightarrow T$$

[8] The alphabet for the set E_ν may be taken to be $\Sigma \cup \{\#\}$ where $\#$ is a new symbol, and we write "(w, w')" as a conventional rendering of the string $w \# w'$.

where $\nu_2|_f(w) = \nu_2(f(w))$ and $T = \texttt{range}(\nu_2 \circ f) \subseteq S_2$. Thus $\nu_2|_f$ is essentially the function composition, $\nu_2 \circ f$, except that the nominal range of $\nu_2 \circ f$ is S_2 instead of T. If f is a recursive function, then we call this operation **recursive restriction**.

We now define "explicitness" by induction: a notation ν is **explicit** if [Base Case] $\nu : \Sigma^* \dashrightarrow S$ is a 1-1 function and S is finite, or [Induction] there exist explicit notations ν_1, ν_2 such that ν is one of the notations

$$\nu_1 \times \nu_2, \quad \nu_1^*, \quad \nu_1|_f$$

where f is recursive. A set S is **explicit** if there exists an explicit notation for S.

Informally, an explicit set is obtained by repeated application of Cartesian product, Kleene star and recursive restriction. Note that Cartesian product, Kleene star and restriction are analogues (respectively) of the Axiom of pairing, Axiom of powers and Axiom of Specification in standard set theory ([19, pp. 9,6,19]). Let us note some applications of recursive restriction: suppose $\nu : \Sigma^* \dashrightarrow S$ is an explicit notation.

- (Change of alphabet) Notations can be based on any alphabet Γ: we can find a suitable recursive function f such that $\nu|_f : \Gamma^* \dashrightarrow S$ is an explicit notation. We can further make $\nu|_f$ a 1-1 function.
- (Identity) The identity function $\nu : \Sigma^* \to \Sigma^*$ is explicit: to see this, begin with $\nu_0 : \Sigma^* \dashrightarrow \Sigma$ where $\nu_0(a) = a$ if $a \in \Sigma$ and $\nu_0(w) = \uparrow$ otherwise. Then ν can be obtained as a recursive restriction of ν_0^*. Thus, Σ^* is an explicit set.
- (Subset) Let T be a subset of S such that $D = \{w : \nu(w) \in T\}$ is recursive. If ι_D is the partial identity function of D, then $\nu|_{\iota_D}$ is an explicit notation for T. We denote $\nu|_{\iota_D}$ more simply by $\nu|_T$.
- (Quotient) Let $\sim$ be an equivalence relation on S, we want a notation for $S/\sim$ (the set of equivalence classes of $\sim$). Consider the set $E = \{(w, w') : \nu(w) \sim \nu(w') \text{ or } \nu(w) = \nu(w') = \uparrow\}$. We say $\sim$ is **recursive relative to** ν if E is a recursive set. Define the function $f : \Sigma^* \to \Sigma^*$ via $f(w) = \min\{w' : (w, w') \in E\}$ (where min is based on any lexicographic order $\leq_{\texttt{LEX}}$ on Σ^*). If E is recursive then f is clearly a recursive function. Then the notation $\nu|_f$, which we denote by

$$\nu/\sim \tag{1}$$

  can be viewed as a notation for $S/\sim$, *provided* we identify $S/\sim$ with a subset of S (namely, each equivalence class of $S/\sim$ is identified with a representative from the class). This identification device will be often used below.

We introduce a normal form for explicit notations. Define a **simple notation** to be one obtained by applications of the Cartesian product and Kleene-star operator to a base case (i.e., to a notation $\nu : \Sigma^* \dashrightarrow S$ that is $1-1$ and S is finite). A **simple set** is one with a simple notation. In other words, simple sets do not need recursive restriction for their definition. A **normal form notation** ν_S for a set S is one obtained as the recursive restriction of a simple notation: $\nu_S = \nu|_f$ for some simple notation ν and recursive function f.

LEMMA 1 (Normal form). *If S is explicit, then it has a normal form notation ν_S.*

Proof. Let $\nu_0 : \Sigma^* \dashrightarrow S$ be an explicit notation for S.
(0) If S is a finite set, then the result is trivial.
(1) If $\nu_0 = \nu_1 \times \nu_2$ then inductively assume the normal form notations $\nu_i = \nu_i'|_{f_i'}$ $(i = 1, 2)$. Let $\nu = \nu_1' \times \nu_2'$ and for $w_i \in \texttt{domain}(\nu_i')$, define f by $f(w_1 \# w_2) = f_1(w_1) \# f_2(w_2) \in S$. Clearly f is recursive and $\nu|_f$ is an explicit notation for S.
(2) If $\nu_0 = \nu_1^*$ then inductively assume a normal form notation $\nu_1 = \nu_1'|_{f_1'}$. Let $\nu = (\nu_1')^*$ and for all $w_j \in \texttt{domain}(\nu_1')$, define $f(w_1 \# w_2 \# \cdots \# w_n) = f_1(w_1) \# f_1(w_2) \# \cdots \# f_1(w_n) \in S$. So $\nu|_f$ is an explicit notation for S.
(3) If $\nu_0 = \nu_1|_{f_1}$, then inductively assume the normal form notation $\nu_1 = \nu_1'|_{f_1'}$. Let $\nu = \nu_1$ and $f = f_1' \circ f_1$. Clearly, $\nu|_f$ is an explicit notation for S. **Q.E.D.**

The following is easily shown using normal form notations:

LEMMA 2. *If ν is explicit, then the sets E_ν and D_ν are recursive.*

In the special case where $S \subseteq \mathbb{N}$ or $S \subseteq \Sigma^*$, we obtain:

LEMMA 3. *A subset of $S \subseteq \mathbb{N}$ is explicit iff S is partial recursive (i.e., recursively enumerable).*

Proof. If S is explicit, then any normal form notation $\nu : \Sigma^* \dashrightarrow S$ has the property that ν is a recursive function, and hence S is recursively enumerable. Conversely, if S is recursively enumerable, it is well-known that there is a total recursive function $f : \mathbb{N} \to \mathbb{N}$ whose range is S. We can use f as our explicit notation for S. **Q.E.D.**

We thus have the interesting conclusion that the halting set K is an explicit set, but *not* by virtue of the canonical (ν_K) or ordered (ν'_K) notations discussed above. Moreover, the complement of K is not an explicit set, confirming that our concept of explicitness is non-trivial (i.e., not every countable set is explicit). Our lemma suggests that explicit sets are analogues of recursively enumerable sets. We could similarly obtain analogues of recursive sets, and a complexity theory of explicit sets can be developed.

The following data structure will be useful for describing computational structures. Let L be any set of symbols. To motivate the definition, think of L as a set of labels for numerical expressions. E.g., $L = \mathbb{Z} \cup \{+, -, \times\}$, and we want to define arithmetic expressions labeled by L.

An **ordered L-digraph** $G = (V, E; \lambda)$ is a directed graph (V, E) with vertex set $V = \{1, \ldots, n\}$ (for some $n \in \mathbb{N}$) and edge set $E \subseteq V \times V$, and a labeling function $\lambda : V \to L$ such that the set of outgoing edges from any vertex $v \in V$ is totally ordered. Such a graph may be represented by a set $\{L_v : v \in V\}$ where each L_v is the **adjacency list** for v, having the form $L_v = (v, \lambda(v); u_1, \ldots, u_k)$ where k is the outdegree of v, and each (v, u_i) is an edge. The total order on the set of outgoing edges from v is specified by this adjacency list. We deem two such graphs $G = (V, E; \lambda)$ and $G' = (V', E'; \lambda')$ to be the same if, up to a renaming of the vertices, they have the same set of adjacency lists (so the identity of vertices is unimportant, but their labels are). Let $\mathcal{DG}(L)$ be the set of all ordered L-digraphs.

LEMMA 4. *Let S, T be explicit sets. Then the following sets are explicit:*
(i) Disjoint union $S \uplus T$,
(ii) Finite power set $\widehat{2}^S$ (set of finite subsets of S),
(iii) The set of ordered S-digraphs $\mathcal{DG}(S)$.

Proof. Let $\underline{x}, \underline{x_i}$ and $\underline{y}$ be representation elements for $x, x_i \in S$, $y \in T$. We use standard encoding tricks. Assume $\#$ is a new symbol.
(i) In disjoint union, tag the representations of S by using $\#\underline{x}$ instead of $\underline{x}$, but $\underline{y}$ is unchanged.
(ii) For the finite power set $\widehat{2}^S$, we apply recursive restriction to the notation ν^* for S^*: define $f(x_1\# \cdots \#_n) = x_{\pi(1)}\# \cdots \# x_{\pi(m)}$ where $x_{\pi(1)} < \cdots < x_{\pi(m)}$ (using any lexicographical ordering $<$ on strings), assuming that $\{x_1, \ldots, x_n\} = \{x_{\pi(1)}, \ldots, x_{\pi(m)}\}$. Then $\nu^*|_f$ is a notation for $\widehat{2}^S$, assuming that we identify $\widehat{2}^S$ with a suitable subset of S^*.
(iii) Recall the representation of an ordered S-digraph above, as a set of adjacency lists, $\{L_v : v \in V\}$. The adjacency lists can be represented via the Kleene star operation, and for the set of adjacency lists we use the finite power set method of (ii). Vertices $v \in \mathbb{N}$ are represented by binary numbers. (Computationally, checking equivalent representations for ordered S-digraphs is highly nontrivial since the graph isomorphism problem is embedded here.) **Q.E.D.**

Convention for Representation Elements. In normal discourse, we prefer to focus on a set S rather than on its representing set T (under some representation $\rho : T \dashrightarrow S$). We introduce an "underbar-convention" to facilitate this. If $x \in S$, we shall write $\underline{x}$ (x-underbar) to denote *some* representing element for x (so $\underline{x} \in T$ is an "underlying representation" of x). This convention makes sense when the representation ρ is understood or fixed. Note that we have already used this convention in the above proof: writing "$\underline{x}$" allows us to acknowledge that it is the representation of x in an unobtrusive way. The fact that "$\underline{x}$" is under-specified (non-unique) is usually harmless.

Example: Dyadic numbers. Let $\mathbb{D} := \mathbb{Z}[\frac{1}{2}] = \{m2^n : m, n \in \mathbb{Z}\}$ denote the set of **dyadic numbers** (or **bigfloats**, in programming contexts). Let $\Sigma_2 = \{0, 1, \bullet, +, -\}$. A string

$$w = \sigma b_{n-1} b_{n-2} \cdots b_k \bullet b_{k-1} \cdots b_0 \tag{2}$$

in Σ_2^* is **proper** if $\sigma \in \{+,-\}$, $n \geq 0$, $k = 0, \ldots, n$ and each $b_j \in \{0,1\}$. The **dyadic notation**

$$\nu_2 : \Sigma_2^* \dashrightarrow \mathbb{D} \tag{3}$$

takes the proper string w in (2) to the number $\nu_2(w) = \sigma 2^{-k} \sum_{i=0}^{n-1} b_i 2^i \in \mathbb{D}$; otherwise $\nu_2(w) = \uparrow$. In order to consider ν_2 as an explicit notation, we will identify $\mathbb{D}$ with a suitable subset $\underline{\mathbb{D}} \subseteq \Sigma_2^*$. $\underline{\mathbb{D}}$ comprises the proper strings (2) with additional properties: $(n > k \Rightarrow b_{n-1} = 1)$, $(k \geq 1 \Rightarrow b_0 = 1)$ and $(n = k = 0 \Rightarrow \sigma = +)$. Thus the strings $+\bullet$, $-1\bullet$, $+10\bullet$, $-11\bullet$, $+\bullet 1$, $-1\bullet 01$, etc, are identified with the numbers $0, -1, 2, -3, 0.5, -1.25$, etc. We can identify $\mathbb{N}$ and $\mathbb{Z}$ as suitable subsets of $\mathbb{D}$, and thus obtain notations for $\mathbb{N}, \mathbb{Z}$ by recursive restrictions of ν_2. All these are explicit notations. We also extend ν_2 to a notation for $\mathbb{Q}$: consider the representation of rational numbers by pairs of integers, $\rho_{\mathbb{Q}} : \mathbb{Z}^2 \dashrightarrow \mathbb{Q}$ where $\rho_{\mathbb{Q}}(m,n) = m/n$ if $n \neq 0$, and $\rho_{\mathbb{Q}}(m,0) = \uparrow$. This induces an equivalence relation $\sim$ on $\mathbb{Z}^2$ where $(m,n) \sim (m',n')$ iff $\rho_{\mathbb{Q}}(m,n) \equiv \rho_{\mathbb{Q}}(m',n')$. We then obtain a notation for $\mathbb{Q}$ by the composition $\rho_{\mathbb{Q}} \circ (\nu_2 \times \nu_2)$. This example illustrates the usual way in which representations ρ of mathematical domains are built up by successive composition of representations, $\rho = \rho_1 \circ \rho_2 \circ \cdots \circ \rho_k$ $(k \geq 1)$. If ρ_k is a notation, then ρ is also a notation. Although ρ may not be an explicit notation, it can be converted into an explicit notation by a natural device. E.g., $\rho_{\mathbb{Q}} \circ (\nu_2 \times \nu_2)$ is not an explicit notation for $\mathbb{Q}$. To obtain an explicit notation for $\mathbb{Q}$, we use the quotient notation $(\nu_2 \times \nu_2)/\sim$, as described in (1). This required an identification of $\mathbb{Q}$ with a subset of $\mathbb{Z}^2$. In fact, we typically identify $\mathbb{Q}$ with the subset of $\mathbb{Z}^2$ comprising relatively prime pairs $(p,q) \in \mathbb{Z}^2$ where $q > 0$. We next formalize this procedure.

Suppose we want to show the explicitness of a set S, and we have a "natural" representation $\rho : T \dashrightarrow S$. For instance, $\rho_{\mathbb{Q}}$ is surely a "natural" representation of $\mathbb{Q}$. We proceed by choosing an explicit notation $\nu : \Sigma^* \dashrightarrow T$ for T. Then

$$\rho \circ \nu$$

is a notation for S, but not necessarily explicit. Relative to ρ, we say that the set S is **naturally identified in** T if we identify each $x \in S$ as some element of the set $\rho^{-1}(x)$. Under this identification, we have $S \subseteq T$. Moreover, the representation ρ is the identity function when its nominal domain is restricted to S. The following lemma then provides the condition for concluding that S is an explicit set.

LEMMA 5. *Let $\rho : T \dashrightarrow S$ be a representation of S. Suppose $\nu : \Sigma^* \dashrightarrow T$ is an explicit notation such that $\rho \circ \nu : \Sigma^* \dashrightarrow S$ is a recursive notation. Then the set S is an explicit set under some natural identification of S in T. In fact, there is recursive function $f : \Sigma^* \dashrightarrow \Sigma^*$ such that $\nu|_f : \Sigma^* \dashrightarrow S$ is an explicit notation.*

Explicit Subsets. So far, set intersection $S \cap T$ and union $S \cup T$ are not among the operations we considered. To discuss these operations, we need a "universal set", taken to be another explicit set U.

Let U be a ν-explicit set where $\nu : \Sigma^* \dashrightarrow U$. We call[9] $S \subseteq U$ a **(partially) ν-explicit subset** of U if the set $\{w \in \Sigma^* : \nu(w) \in S\}$ is (partially) recursive. By definition, S is an explicit subset of U iff U is an **explicit superset** of S.

LEMMA 6. *Let U be ν-explicit, and S and T are ν-explicit subsets of U.*
(i) The sets S, $U \setminus S$, $S \cup T$ and $S \cap T$ are all explicit sets.
(ii) The set of ν-explicit subsets of U is closed under the Boolean operations (union, intersection, complement).
(iii) The set of partially ν-explicit subsets of U is closed under union.
(iv) There exists U and a partially ν-explicit subset $S \subseteq U$ such that $U \setminus S$ is not an explicit set.

Proof. To show (iv), we let $U = \Sigma^*$, where ν is the identity function, and let S be the halting set. **Q.E.D.**

Abstract and Concrete Sets. Before concluding this section, we make two remarks on the interplay between abstract and concrete sets. For this discussion, let us agree to call a set **concrete** if it is a subset of some Σ^*; other kinds of set are said to be **abstract**. Following Turing, algorithms can only treat concrete

[9] In this paper, any appearance of the parenthetical "(partially)" conveys two parallel statements: one with all occurrences of "(partially)" removed, and another with "partially" inserted.

sets; but in mathematics, we normally treat abstract sets like $\mathbb{Q}, \mathbb{R}, Hom(A,B), SO(3)$, etc. It is seen from the above development that if we want explicit notations for abstract sets, we must ultimately identify them with a suitable subset of a simple set (cf. (1)). Thus, the set of all subsets of strings is our "canonical universe" for abstract sets. In practice, making such identifications of abstract sets is not an arbitrary process (it is not enough to obtain just any bijection). Abstract sets in reality have considerable structure and relation with other abstract sets, and these must be preserved and made available for computation. Our explicit sets can encode such information, but this is is not formalized. In other words, our sets are conceptually "flat". Exploring structured sets is a topic for future work. Without formalizing these requirements, we must exercise judgment in making such identifications on a case by case basis, but typically they are non-issues as in, e.g., $\mathbb{Q}$ (above), ideals $\mathcal{ID}_n$ (Lemma 8) and real closure $\overline{F}$ (Theorem 12). The second remark concerns the manipulation of an abstract set S through its notation ν. Strictly speaking, all our development could have been carried out by referring only to the equivalence relation E_ν, and never mentioning S. But this approach would be tedious and unnatural for normal human comprehension. Being able to talk directly about S is more efficient. For this practical[10] reason that we prefer to talk about S, and (like the underbar-convention) only allude to the contingent E_ν.

3 Explicit Algebraic Structures

In this section, we extend explicit sets to explicit algebraic structures such as rings and fields. To do this, we need to introduce explicit functions.

Function representations. To represent functions, we assume a representation of its underlying sets. Let $\rho : T \dashrightarrow S$ be a representation. If $f : S^2 \dashrightarrow S$ is a function, then the function $\underline{f} : T \times T \dashrightarrow T$ is called a **ρ-representation** of f if for all $x, y \in T$,

$$\rho(\underline{f}(x,y)) \equiv f(\rho(x), \rho(y)).$$

In case ρ is a notation, we call $\underline{f}$ a **ρ-notation** for f.

We say $f : S^2 \dashrightarrow S$ is **ν-explicit** (or simply "explicit") if S is ν-explicit and there exists a recursive ν-notation $\underline{f}$ for f. Although we never need to discuss "partially explicit sets", we will need "partially explicit functions": we say f is **partially ν-explicit** (or simply "partially explicit") if S is ν-explicit and there is a ν-notation $\underline{f}$ for f which is partial recursive.

These concepts are naturally extended to general k-ary functions, and to functions whose range and domain involve different sets. E.g., if $f : S \dashrightarrow T$ where S is ν-explicit and T is ν'-explicit, we can talk about f being (ν, ν')-explicit. These concepts also apply to the special case of predicates: we define a **predicate on** S to be any partial function $R : S \dashrightarrow V$ where V is any finite set. Usually $|V| = 2$, but geometric predicates often have $|V| = 3$ (e.g., $V = \{IN, OUT, ON\}$).

An **(algebraic) structure** is a pair (S, Ω) where S is a set (the carrier set) and Ω is a set of predicates and algebraic operations on S. By an **(algebraic) operation** on S we mean any partial function $f : S^k \dashrightarrow S$, for some $k \geq 0$ (called the **arity** of f).

As an example, and one that is used extensively below, an **ordered ring** S is an algebraic structure with $\Omega = \{+, -, \times, 0, 1, \leq\}$ such that $(S, +, -, \times, 0, 1)$ is a ring[11] and $\leq$ is a total order on S with the following properties: the ordering defines a **positive set** $P = \{x \in R : x > 0\}$ that is closed under $+$ and $\times$, and for all $x \in S$, exactly one of the following conditions is true: $x = 0$ or $x \in P$ or $-x \in P$. See [40, p. 129].

Ordered rings are closely related to another concept: a ring R is **formally real** if $0 \notin R^{(2)}$ where $R^{(2)}$ is the set of all sums of non-zero squares. When the rings are fields or domains, etc, we may speak of ordered fields, formally real domains, etc. Note that ordered rings are formally real, and formally real rings must be domains and can be extended into formally real fields. Conversely, formally real fields can be extended into an ordered field (its real constructible closure) [40, Theorem 5.6, p. 134]. Clearly, formally real fields have characteristic 0.

The key definition is this: an algebraic structure (S, Ω) is **ν-explicit** if its carrier set is ν-explicit, and each $f \in \Omega$ is ν-explicit. Thus we may speak of explicit rings, explicit ordered fields, etc.

[10] If not for some deeper epistemological reason.

[11] All our rings are commutative with unit 1.

We must discuss substructures. By a **substructure** of (S, Ω) we mean (S', Ω') such that $S' \subseteq S$, there is a bijection between Ω' and Ω, and each $f' \in \Omega'$ is the restriction to S' of the corresponding operation or predicate $f \in \Omega$. Thus, we may speak of subfields, subrings, etc. If (S, Ω) is ν-explicit, then (S', Ω') is a **ν-explicit substructure** of (S, Ω) if S' is a ν-explicit subset of S and (S', Ω') is a a substructure of (S, Ω). Thus, we have explicit subrings, explicit subfields, etc. If (S', Ω') is an explicit substructure of (S, Ω), we call (S, Ω) an **explicit extension** of (S', Ω').

The following shows the explicitness of some standard algebraic constructions:

LEMMA 7. *Let ν_2 be the normal form notation for $\mathbb{D} = \mathbb{Z}[\frac{1}{2}]$ in (3).*
(i) $\mathbb{D}$ is a ν_2-explicit ordered ring.
(ii) $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{D}$ are ν_2-explicit ordered subrings.
(iii) If D is an explicit domain, then the quotient field $Q(D)$ is an explicit ring extension of D.
(iv) If R is an explicit ring, then the polynomial ring $R[X]$ is an explicit ring extension of R.
(v) If F is an explicit field, then any simple algebraic extension $F(\theta)$ is an explicit field extension of F.

Proof. (i)-(ii) are obvious. The constructions (iii)-(v) are standard algebraic constructions; these constructions can be implemented using operations of explicit sets. Briefly:
(iii) The standard representation of $Q(D)$ uses D^2 as representing set. This is a direct generalization of the construction above, which gave an explicit notation ν for $\mathbb{Q}$ $(= Q(\mathbb{Z}))$ starting from an explicit notation for $\mathbb{Z}$. Now, we need to further verify that the field operations of $Q(D)$ are ν-explicit.
(iv) The standard representation of $R[X]$ uses R^* (Kleene star) as representing set: $\rho : R^* \dashrightarrow R[X]$. Since R is explicit, so is R^*, and hence $R[X]$. All the polynomial ring operations are also recursive relative to this notation for $R[X]$.
(v) Assume θ is the root of an irreducible polynomial $p(X) \in F[X]$ of degree n. The elements of $F(\theta)$ can be directly represented by elements of F^n (n-fold Cartesian product): thus, if ν is an explicit notation for F, then ν^n is an explicit notation of $F^n = F(\theta)$. The ring operations of $F(\theta)$ are reduced to polynomial operations modulo $p(X)$, and division is reduced to computing inverses using the Euclidean algorithm. It is easy to check that these operations are ν^n-explicit. **Q.E.D.**

This lemma is essentially a restatement of corresponding results in [18]. It follows that standard algebraic structures such as $\mathbb{Q}[X_1, \ldots, X_n]$ or algebraic number fields are explicit. Clearly, many more standard constructions can be shown explicit (e.g., explicit matrix rings). The next lemma uses constructions whose explicitness are less obvious: let $\mathcal{ID}_n(F)$ denote the set of all ideals of $F[X_1, \ldots, X_n]$ where F is a field. For ideals $I, J \in \mathcal{ID}_n(F)$, we have the ideal operations of sum $I+J$, product IJ, intersection $I \cap J$, quotient $I : J$, and radical $\sqrt{I}$ [40, p. 25]. These operations are all effective, for instance, using Gröbner basis algorithms [40, chap. 12].

LEMMA 8. *Let F be an explicit field. Then the set $\mathcal{ID}_n(F)$ of ideals has an explicit notation ν, and the ideal operations of sum, product, intersection, quotient and radical are ν-explicit.*

Proof. From Lemma 7(iv), $F[X_1, \ldots, X_n]$ is explicit. From Lemma 4(ii), the set S of finite subsets of $F[X_1, \ldots, X_n]$ is explicit. Consider the map $\rho : S \to \mathcal{ID}_n(F)$ where $\rho(\{g_1, \ldots, g_m\})$ is the ideal generated by $g_1, \ldots, g_m \in F[X_1, \ldots, X_n]$. By Hilbert's basis theorem [40, p. 302], ρ is an onto function, and hence a representation. If $\nu : \Sigma^* \dashrightarrow S$ is an explicit notation for S, then $\rho \circ \nu$ is a notation for $\mathcal{ID}_n(F)$. To show that this notation is explicit, it is enough to show that the equivalence relation E_ρ is decidable (cf. (1)). This amounts to checking if two finite sets $\{f_1, \ldots, f_\ell\}$ and $\{g_1, \ldots, g_m\}$ of polynomials generate the same ideal. This can be done by computing their Gröbner bases (since such operations are all rational and thus effective in an explicit field), and seeing each reduces the other set of polynomials to 0. Let S/E_ρ denote the equivalence classes of S; by identifying S/E_ρ with the set $\mathcal{ID}_n(F)$, we obtain an explicit notation for $\mathcal{ID}_n(F)$, $\nu/E_\rho : \Sigma^* \dashrightarrow S/E_\rho$. The (ν/E_ρ)-explicitness of the various ideal operations now follows from known algorithms, using the notation (ν/E_ρ). **Q.E.D.**

Well-ordered sets. Many algebraic constructions (e.g., [38, chap. 10]) are transfinite constructions, e.g., the algebraic closure of fields. The usual approach for showing closure properties of such constructions depends on the well-ordering of sets (Zermelo's theorem), which in turn depends on the Axiom of Choice (e.g., [19] or [38, chap. 9]). Recall that a strict total ordering $<$ of a set S is a well-ordering if every non-empty

subset of S has a least element. In explicit set theory, we can replace such axioms by theorems, and replace non-constructive constructions by explicit ones.

LEMMA 9. *A ν-explicit set is well-ordered. This well-ordering is ν-explicit.*

Proof. Let $\nu : \Sigma^* \dashrightarrow S$ be an explicit notation for S. Now Σ^* is well-ordered by any lexicographical order $\leq_{\mathtt{LEX}}$ on strings. This induces a well-ordering $\leq_\nu$ on the elements $x, y \in S$ as follows: let $w_x := \min\{w \in \Sigma^* : \nu(w) = x\}$. Define $x \leq_\nu y$ if $w_x \leq_{\mathtt{LEX}} w_y$. The predicate $\leq_\nu$ is clearly ν-explicit. Moreover it is a well-ordering. **Q.E.D.**

The proof of Theorem 12 below depends on such a well-ordering.

Expressions. Expressions are basically "universal objects" in the representation of algebraic constructions.

Let $\widehat{\Omega}$ be a (possibly infinite) set of symbols for algebraic operations, and $k : \widehat{\Omega} \to \mathbb{N}$ assigns an "arity" to each symbol in $\widehat{\Omega}$. The pair $(\widehat{\Omega}, k)$ is also called a signature. Suppose Ω is a set of operations defined on a set S. To prove the closure of S under the operations in Ω, we consider "expressions" over $\widehat{\Omega}$, where each $\widehat{g} \in \widehat{\Omega}$ is interpreted by a corresponding $g \in \Omega$ and $k(\widehat{g})$ is the arity of g. To construct the closure of S under Ω, we will use "expressions over $\widehat{\Omega}$" as the representing set for this closure.

Let $\widehat{\Omega}^{(k)}$ denote the subset of $\widehat{\Omega}$ comprising symbols with arity k. Recall the definition of the set $\mathcal{DG}(\widehat{\Omega})$ of ordered $\widehat{\Omega}$-digraphs. An **expression over** $\widehat{\Omega}$ is a digraph $G \in \mathcal{DG}(\widehat{\Omega})$ with the property that (i) the underlying graph is acyclic and has a unique source node (the root), and (ii) the outdegree of a node v is equal to the arity of its label $\lambda(v) \in \widehat{\Omega}$. Let $Expr(\widehat{\Omega}, k)$ (or simply, $Expr(\widehat{\Omega})$) denote the set of expressions over $\widehat{\Omega}$.

LEMMA 10. *Suppose $\widehat{\Omega}$ is a ν-explicit set and the function $k : \widehat{\Omega} \to \mathbb{N}$ is*[12] *ν-explicit. Then the set $Expr(\widehat{\Omega})$ of expressions is an explicit subset of $\mathcal{DG}(\widehat{\Omega})$.*

Proof. The set $\mathcal{DG}(\widehat{\Omega})$ is explicit by Lemma 4(iii). Given a digraph $G = (V, E; \lambda) \in \mathcal{DG}(\widehat{\Omega})$, it is easy to algorithmically check properties (i) and (ii) above in our definition of expressions. **Q.E.D.**

Universal Real Construction. A fundamental result of field theory is Steinitz's theorem on the existence and uniqueness of algebraic closures of a field F [38, chap. 10]. In standard proofs, we only need the well-ordering principle. To obtain the "explicit version" of Steinitz's theorem, it is clear that we also need F to be explicit. But van der Waerden pointed out that this may be insufficient: in general, we need another explicitness assumption, namely the ability to factor over $F[X]$ (see [18]). Factorization in an explicit UFD (unique factorization domain) such as $F[X]$ is equivalent to checking irreducibility [18, Theorem 4.2].

If F is a formally real field, then a **real algebraic closure** $\overline{F}$ of F is an algebraic extension of F that is formally real, and such that no proper algebraic extension is formally real. Again $\overline{F}$ exists [40, chap. 5], and is unique up to isomorphism. Our goal here is to give the explicit analogue of Steinitz's theorem for real algebraic closure.

If $p, q \in F[X]$ are polynomials, then we consider the operations of computing their remainder $p \,\mathbf{mod}\, q$, their quotient $p \,\mathbf{quo}\, q$, their gcd $\mathtt{GCD}(p, q)$, their resultant $\mathtt{resultant}(p, q)$, the derivative $\frac{dp}{dX}$ of p, the square-free part $\mathtt{sqfree}(p)$ of p, and the Sturm sequence $\mathrm{Sturm}(p)$ of p. Thus $\mathrm{Sturm}(p)$ is the sequence $(p_0, p_1, \ldots, p_k)$ where $p_0 = p$, $p_1 = \frac{dp}{dX}$, and $p_{i+1} = p_{i-1} \,\mathbf{mod}\, p_i$ $(i = 1, \ldots, k)$, and $p_{k+1} = 0$. These are all explicit in an explicit field:

LEMMA 11. *If F is a ν-explicit field, and $p(X), q(X) \in F[X]$, then the following operations are*[13] *ν-explicit:*

$$p \,\mathbf{mod}\, q, \quad p \,\mathbf{quo}\, q, \quad \mathtt{GCD}(p, q), \quad \frac{dp}{dX}, \quad \mathtt{resultant}(p, q), \quad \mathtt{sqfree}(p), \quad \mathrm{Sturm}(p).$$

[12] Strictly speaking, k is (ν, ν')-explicit where ν' is the notation for $\mathbb{N}$.

[13] Technically, these operations are ν'-explicit where $F[X]$ is an ν'-explicit set, and ν' is derived from ν using the above standard operators.

Proof. Let $prem(p,q)$ and $pquo(p,q)$ denote the pseudo-remainder and pseudo-quotient of $p(X), q(X)$ [40, Lemmas 3.5, 3.8]. Both are polynomials whose coefficients are determinants in the coefficients of $p(X)$ and $q(X)$. Hence $prem(p,q)$ and $pquo(p,q)$ are explicit operations. The leading coefficients of $prem(p,q)$ and $pquo(p,q)$ can be detected in an explicit field. Dividing out by the leading coefficient, we can obtain $p \,\mathbf{mod}\, q$ and $p \,\mathbf{quo}\, q$ from their pseudo-analogues. Similarly, $\texttt{GCD}(p,q)$ and $\texttt{resultant}(p,q)$ can be obtained via subresultant computations [40, p. 90ff]. Clearly, differentiation $\frac{dp}{dX}$ is a ν-explicit operation. We can compute $\texttt{sqfree}(p)$ as $p/\texttt{GCD}(p, dp/dX)$. Finally, we can compute the Sturm sequence $\text{Sturm}(p)$ because we can differentiate and compute $p \,\mathbf{mod}\, q$, and can test when a polynomial is zero. **Q.E.D.**

Some common predicates are easily derived from these operations, and they are therefore also explicit predicates: (a) $p|q$ (p divides q) iff $p \,\mathbf{mod}\, q = 0$. (b) p is squarefree iff $\texttt{sqfree}(p) = p$.

Let F be an ordered field. Given $p \in F[X]$, an interval I is an **isolating interval** of p in one of the following two cases: (i) $I = [a,a]$ and $f(a) = 0$ for some $a \in F$, (ii) $I = (a,b)$ where $a, b \in F$, $a < b$, $p(a)p(b) < 0$, and the Sturm sequence of p evaluated at a has one more sign variation than the Sturm sequence of p evaluated at b. It is clear that an isolating interval uniquely identifies a root α in the real algebraic closure of F. Such an α is called a **real root** of p. In case p is square-free, we call the pair (p, I) an **isolating interval representation** for α. We may now define the operation $\texttt{Root}_k(a_0, \ldots, a_n)$ ($k \geq 1$, $a_i \in F$) that extracts the kth largest real root of the polynomial $p(X) = \sum_{i=0}^{n} a_i X^i$. This operation is undefined in case $p(X)$ has less than k real roots. For the purposes of this paper, we shall define the **real algebraic closure** of F, denoted $\overline{F}$, to be the smallest ordered field that is an algebraic extension of F and that is closed under the operation $\texttt{Root}_k(a_0, \ldots, a_n)$ for all $a_0, \ldots, a_n \in \overline{F}$ and $k \in \mathbb{N}$. For other characterizations of real algebraic closures, see e.g., [40, Theorem 5.11].

THEOREM 12. *Let F be an explicit ordered field. Then the real algebraic closure $\overline{F}$ of F is explicit. This field is unique up to F-isomorphism (isomorphism that leaves F fixed).*

Unlike Steinitz's theorem [38, chap. 10], this result does not need the Axiom of Choice; and unlike the explicit version of Steinitz's theorem [18], it does do not need factorization in $F[X]$. But the ordering in F must be explicit.

Proof. For simplicity in this proof, we will assume the existence of $\overline{F}$ (see [40]). So our goal is to show its explicitness, i.e., we must show an explicit notation ν for $\overline{F}$, and show that the field operations as well as $\texttt{Root}_k(a_0, \ldots, a_n)$ are ν-explicit. Consider the set

$$\widehat{\Omega} := F \cup \{+, -, \times, \div\} \cup \{\texttt{Root}_k : n \in \mathbb{N}, 1 \leq k \leq n\}$$

of operation symbols. The arity of these operations are defined as follows: the arity of $x \in F$ is 0, arity of $g \in \{+, -, \times, \div\}$ is 2, and arity of $\texttt{Root}_k$ is $n+1$. It is easy to see that $\widehat{\Omega}$ is explicit, and hence $Expr(\widehat{\Omega})$ is explicit. Define a natural evaluation function,

$$\text{Eval} : Expr(\widehat{\Omega}) \dashrightarrow \overline{F}. \tag{4}$$

Let e be an expression. We assign $\texttt{val}(u) \in \overline{F}$ to each node u of the underlying DAG of e, in bottom-up fashion. Then $\text{Eval}(e)$ is just the value of the root. If any node has an undefined value, then $\text{Eval}(e) = \uparrow$. The leaves are assigned constants $\texttt{val}(u) \in F$. At an internal node u labeled by a field operation $(+, -, \times, \div)$, we obtain $\texttt{val}(u)$ as the result of the corresponding field operation on elements of $\overline{F}$. Note that a division by 0 results in $\texttt{val}(u) = \uparrow$. Similarly, if the label of u is $\texttt{Root}_k$ and its children are $u_0, \ldots, u_n$ (in this order), then $\texttt{val}(u)$ is equal to the kth largest real root (if defined) of the polynomial $\sum_{j=0}^{n} \texttt{val}(u_j) X^j$. We notice that the evaluation function (4) is onto, and hence is a representation of the real algebraic closure $\overline{F}$. Since $Expr(\widehat{\Omega})$ is an explicit set, Eval is a notation of $\overline{F}$. It remains to show that Eval is an explicit notation.

To conclude that $\overline{F}$ is an explicit set via the notation (4), we must be able to decide if two expressions e, e' represent the same value. By forming the expression $e - e'$, this is reduced to deciding if the value of a proper expression e is 0. To do this, assume that for each expression e, we can either determine that it is improper or else we can compute an isolating interval representation $(P_e(X), I_e)$ for its value $\texttt{val}(e)$. To determine if $\texttt{val}(e) = 0$, we first compute the isolating interval representation (P_e, I_e). If $P_e(X) = \sum_{i=0}^{n} a_i X^i$, and if $\texttt{val}(e) \neq 0$, then Cauchy's lower bound [40, Lem. 6.7] holds: $|\texttt{val}(e)| > |a_0|/(|a_0| + \max\{|a_i| : i = 1, \ldots, n\})$.

Let $w(I) = b - a$ denote the width of an interval $I = (a, b)$ or $I = [a, b]$. Therefore, if

$$w(I_e) < B_0 := \frac{|a_0|}{|a_0| + \max\{|a_i| : i = 1, \ldots, n\}}, \tag{5}$$

we can decide whether $\texttt{val}(e) = 0$ as follows: $\texttt{val}(e) = 0$ iff $0 \in I_e$. To ensure (5), we first compute B_0 and then we use binary search to narrow the width of I_e: each binary step on the current interval $I' = (a', b')$ amounts to computing $m = (a' + b')/2$ and testing if $P_e(m) = 0$ (which is effective in F). If so, $\texttt{val}(e) = m$ and we can tell if $m = 0$. Otherwise, we can replace I' by (a', m) or (m, b'). Specifically, we choose (a', m) if $P_e(a')P(m) < 0$ and otherwise choose (m, b'). Thus the width of the interval is halved. We repeat this until the width is less than B_0.

Given $P(X) \in F[X]$, we need to compute a complete set of isolating intervals for all the real roots of $P(X)$ using Sturm sequences. This is well known if $F \subseteq \mathbb{R}$, but it is not hard to see that everything extends to explicit ordered fields. Briefly, three ingredients are needed: (i) The Sturm sequence $\text{Sturm}_P(X)$ of $P(X)$ is $\text{Sturm}_P(X) := (P_0, P_1, \ldots, P_h)$ where $P_0 = P(X)$, $P_1 = dP(X)/dX$, and $P_{i+1} = P_{i-1} \bmod P_i$ $(i = 1, \ldots, h)$, and $P_{h+1} = 0$. To compute $\text{Sturm}_P(X)$, we need to compute polynomial remainders for polynomials in $F[X]$, and be able to detect zero coefficients (so that we know the leading coefficients). (ii) We need an upper bound B_1 on the magnitude of all roots. The Cauchy bound may be used: choose $B_1 = 1 + (\max_{i=0}^{n-1} |a_i|)/|a_n|$ where $a_0, \ldots, a_n$ are coefficients of the polynomial [40, p. 148]. (iii) Let $V_P(a)$ denote the number of sign variations of the Sturm sequence of P evaluated at a. The usual theory assumes that $P(a)P(b) \neq 0$. In this case, the number of distinct real roots of $P(X)$ in an interval (a, b) is given by $V_P(a) - V_P(b)$. But in case $P(a) = 0$ and/or $P(b) = 0$, we need to compute $V_P(a^+)$ and/or $V_P(b^-)$. As noted in [16], if P is square-free, then for all $a, b \in F$, we have $V_P(a^+) = V_P(a)$ and $V_P(b^-) = \delta(P(b)) + V_P(b)$ where $\delta(x) = 1$ if $x = 0$ and $\delta(x) = 0$ otherwise. This computation uses only ring operations and sign determination.

We now return to our main problem, which is to compute, for given expression e, an isolating interval representation of $\texttt{val}(e)$ or determine that e is improper. The algorithm imitates the preceding bottom-up assignment of values $\texttt{val}(u)$ to each node u, except that we now compute an isolating interval representation of $\texttt{val}(u)$ at each u. Let this isolating interval representation be $(P_u(X), I_u)$. If u is a leaf, $P_u = X - \texttt{val}(u)$, and $I_u = [\texttt{val}(u), \texttt{val}(u)]$. Inductively, we have two possibilities:

(I) Suppose $\lambda(u)$ is a field operation $\diamond \in \{+, -, \times, \div\}$ and the children of u are v, w. Recursively, assume $\texttt{val}(v), \texttt{val}(w)$ are defined and we have computed the isolating interval representations (P_v, I_v) and (P_w, I_w). In case $\diamond$ is division, we next check if $\texttt{val}(w) = 0$. If so, $\texttt{val}(u)$ is undefined. If not, then $P_u(X)$ may be given by the square-free part of the following resultants (respectively):

$$\begin{aligned} v \pm w: &\quad \texttt{res}_Y(P_v(Y), P_w(X \mp Y)) \\ v \times w: &\quad \texttt{res}_Y(P_v(Y), Y^{\deg(P_w)} P_w(X/Y)) \\ v/w: &\quad \texttt{res}_Y(P_v(Y), X^{\deg(P_w)} P_w(Y/X)). \end{aligned}$$

See [40, p. 158]; the last case can be deduced from the reciprocal formula there. If the operation is division, we first use the binary search procedure above to narrow the width of I_w until either $0 \notin I_w$ or we detect that $\texttt{val}(w) = 0$ i.e., $w(I_w) < B_0$ and $0 \in I_w$. In the latter case, v/w is not defined. Next, tentatively set $I_u = I_v \diamond I_w$ (using interval arithmetic) and test if it is an isolating interval of $P_u(X)$. This is done by using Sturm sequences: by evaluating the Sturm sequence at the endpoints of I_u, we can tell if I_u is isolating. If not, we will half the intervals I_v and I_w using bisection search as above, and repeat the test. It is clear that this process terminates.

(II) Suppose $\lambda(u)$ is $\texttt{Root}_i(u_0, \ldots, u_n)$. To compute $P_u(X) \in F[X]$, we recall that by the theory of elementary symmetric functions, there is a polynomial $P(X) \in F[X]$ of degree at most $D = \prod_{j=0}^{n} \deg(P_{u_j})$ such that $Q(X)|P(X)$ and $Q(X) = \sum_{j=0}^{n} \texttt{val}(u_j) X^j$ [40, Proof of Theorem 6.24]. We can construct an expression over the ring $\overline{F}[X]$ to represent $Q(X)$ (since the coefficient of $Q(X)$ are elements of $\overline{F}$). Moreover, $F[X]$ is an explicit polynomial ring, so we can systematically search for a $P(X) \in F[X] \subseteq \overline{F}[X]$ that is divisible by $Q(X)$. We must show that the predicate $Q(X)|P(X)$ is effective. This is equivalent to checking if $P(X) \bmod Q(X) = 0$. As noted in the proof of Lemma 11, $R(X) := P(X) \bmod Q(X)$ is a polynomial whose coefficients are determinants in the coefficients of $P(X), Q(X)$. Using the resultant techniques in (I), we can therefore compute isolating interval representations of the coefficients of $R(X)$. Thus $R(X) = 0$ iff all

the coefficients of $R(X)$ vanishes, a test that is effective. (Note: this test is a kind of "bootstrap" since we are using isolating interval representations to determine the isolating interval representation of u.) Finally, from $P(X)$, we compute its square-free part P_u.

Next, we must compute an isolating interval I_u of P_u for the ith largest root of the polynomial $Q(X)$. Although we can isolate the real roots of P_u, we must somehow identify the root that corresponds to the ith largest root of $Q(X)$. The problem is that $Q(X) \in \overline{F}[X] \setminus F[X]$, and so we cannot directly use the Sturm method of (I) above. Nevertheless, it is possible to use the Sturm method in a bootstrap manner, by exploiting the isolating interval representation (similar to checking if $Q(X)|P(X)$ above). Indeed, all the operations in Lemma 11 can be implemented in this bootstrap manner. Once we have isolated these roots of $Q(X)$, we can continue to narrow the intervals as much as we like. In particular, we must narrow the ith largest root of $Q(X)$ until it is also an isolating interval for $P_u(X)$. This completes our proof. **Q.E.D.**

REMARK: An alternative method to isolate all the real roots of $Q(X) \in \overline{F}[X]$ may be based on a recent result of Eigenwillig et al [1]. They showed an algorithm to isolate the real roots of square-free polynomials with real coefficients where the coefficients are given as *potentially infinite bit streams*. To apply this algorithm we must first make $Q(X)$ square-free, but this can be achieved using the boot-strap method.

4 Real Approximation

We now address the problem of real computation. Our approach is a further elaboration of [41].

All realizable computations over abstract mathematical domains S are ultimately reduced to manipulation of their representations. If we accept Turings' general analysis of computation, then these representations of S are necessarily notations. But it is impossible to provide notations if S is an uncountable set such as $\mathbb{R}$. There are two fundamental positions to take with respect to this dilemma.

(A) One position insists that a theory of real computation must be able to compute with all of $\mathbb{R}$ from the start. Both the (Polish) analytic school and the algebraic school adopt this line. The analytic school proceeds to generalize the concept of computation to handle representations (of real number, real functions, real operators, etc). So we must generalize Turing machines to handle such non-notation representations: two examples of such generalizations are oracle machines [23] and TTE machines [39]. The algebraic school chooses an abstract computational model that directly manipulates the real numbers, in effect ignoring all representation issues. The decision to embrace all of $\mathbb{R}$ from the start exacts a heavy toll. We believe that the standard criticisms of both schools stem from this fundamental decision: the theory is either unrealizable (algebraic school) or too weak (analytic school, which treats only continuous functions). The resulting complexity theory is also highly distorted unless it is restricted in some strong way. In the algebraic school, one approach is to focus on the "Boolean part" (e.g., [3]). Another is to analyze complexity as a function of condition number [6]. In the analytic school, one focuses[14] mainly on "precision complexity", which is essentially local complexity or complexity at a point.

(B) The alternative position is to accept that there will be real numbers that are simply "inaccessible" for computation. This phenomenon seems inevitable. Once we accept this principle, there is no reason to abandon or generalize Turing's fundamental analysis – for the "accessible reals", we can stick to computations over notations (using standard Turing machines). The Russian branch of the analytic school [39, Chap. 9] takes this approach, by identifying the accessible reals with the computable reals (see below; also Spreen [37]). A real number is now represented by its Gödel numbers (names of programs for computing its Cauchy sequences). Alas, this view also swallows too much in one gulp, again leading to a distorted complexity theory. Our approach [41] takes position (B), but adopts a more constructive view about which real numbers ought to be admitted from the start. Nevertheless, the set of reals that can be studied by our approach is not fixed in advance (see below).

[14] That is, the only useful parameter in complexity functions is the precision parameter p. E.g., for $f : \mathbb{R} \to \mathbb{R}$, the main complexity function we can associate with f is $T(p)$, giving the worst number of steps of an oracle Turing machine for approximating p-bits of $f(x)$ for all $x \in \mathbb{R}$. There is no natural way to use the real value x (or $|x|$) as a complexity in computing $f(x)$. Thus Ko [23, p. 57] defines the complexity function $T(x, p)$ as the time to compute $f(x)$ to p-bits of absolute accuracy. But the x parameter is instantly factored out by considering uniform time complexity, and never used in actual complexity results. The real parameter x fails to behave properly as a complexity parameter: the function $T(x, p)$ is not monotonic in x (for $x > 0$). Even if x is rational, the monotonicity property fails. To be concrete, suppose $T(x, p)$ be the time to compute a p-bit approximation to $\sqrt{x}$. The inequality $T(x, p) \leq T(2, p)$ fails as badly as we like, by choosing $x = (n + 1)/n$ as $n \to \infty$.

Base Reals. We begin with a set of "base reals" that is suitable for approximating other real numbers. Using the theory of explicit algebraic structures, we can now give a succinct definition (cf. [41]): a subset $\mathbb{F} \subseteq \mathbb{R}$ is called a **ring of base reals** if $\mathbb{F}$ is an explicit ordered ring extension of the integers $\mathbb{Z}$, such that $\mathbb{F}$ is dense in $\mathbb{R}$. Elements of $\mathbb{F}$ are called **base reals**.

The rational numbers $\mathbb{Q}$, or the dyadic (or bigfloat) numbers $\mathbb{D} = \mathbb{Z}[\frac{1}{2}]$, or even the real algebraic numbers, can serve as the ring of base reals. Since $\mathbb{F}$ is an explicit ring, we can perform all the ring operations and decide if two base reals are equal. Being dense, we can use $\mathbb{F}$ to approximate any real number to any desired precision. We insist that all inputs and outputs of our algorithms are base reals. This approach reflects very well the actual world of computing: in computing systems, floating point numbers are often called "reals". Basic foundation for this form of real computation goes back to Brent [8, 9]. It is also clear that all practical development of real computation (e.g., [28, 30, 25]), as in our work in EGC, also ultimately depend on approximations via base reals. The choice of $\mathbb{D}$ as the base reals is the simplest: assuming that $\mathbb{F}$ is closed under the map $x \mapsto x/2$, then $\mathbb{D} \subseteq \mathbb{F}$. *In the following, we shall assume this property.* Then we can do standard binary searches (divide by 2), work with dyadic notations, and all the results in [41] extends to our new setting.

Error Notation. We consider both absolute and relative errors: given $x, \widetilde{x}, p \in \mathbb{R}$, we say that $\widetilde{x}$ is an **absolute p-bit approximation** of x if $|\widetilde{x} - x| \leq 2^{-p}$. We say $\widetilde{x}$ is a **relative p-bit approximation** of x if $|\widetilde{x} - x| \leq 2^{-p}|x|$.

The inequality $|\widetilde{x} - x| \leq 2^{-p}$ is equivalent to $\widetilde{x} = x + \theta 2^{-p}$ where $|\theta| \leq 1$. To avoid introducing an explicit variable θ, we will write this in the suggestive form "$\widetilde{x} = x \pm 2^{-p}$". More generally, whenever we use the symbol '$\pm$' in a numerical expression, the symbol $\pm$ in the expression should be replaced by the sequence "$+\theta$" where θ is a real variable satisfying $|\theta| \leq 1$. Like the big-Oh notations, we think of the $\pm$-convention as a variable hiding device. As further example, the expression "$x(1 \pm 2^{-p})$" denotes a relative p-bit approximation of x. Also, write $(x \pm \varepsilon)$ and $[x \pm \varepsilon]$ (resp.) for the intervals $(x - \varepsilon, x + \varepsilon)$ and $[x - \varepsilon, x + \varepsilon]$.

Absolute and Relative Approximation. The ring $\mathbb{F}$ of base reals is used for approximation purposes. So all approximation concepts will depend on this choice. If $f : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$ is[15] a real function, we call a function

$$\widetilde{f} : (S \cap \mathbb{F}^n) \times \mathbb{F} \dashrightarrow \mathbb{F} \tag{6}$$

an **absolute approximation** of f if for all $\mathbf{d} \in S \cap \mathbb{F}^n$ and $p \in \mathbb{F}$, we have $\widetilde{f}(\mathbf{d}, p) =\downarrow$ iff $f(\mathbf{d}) =\downarrow$. Furthermore, when $f(\mathbf{d}) =\downarrow$ then $\widetilde{f}(\mathbf{d}, p) = f(\mathbf{d}) \pm 2^{-p}$. We can similarly define what it means for $\widetilde{f}$ to be a **relative approximation** of f. Let

$$\mathcal{A}_f, \qquad \mathcal{R}_f$$

denote the set of all absolute, respectively relative, approximations of f. If $\widetilde{f} \in \mathcal{A}_f \cup \mathcal{R}_f$, we also write "$\widetilde{f}(\mathbf{d})[p]$" instead of $\widetilde{f}(\mathbf{d}, p)$ to distinguish the **precision parameter** p. We remark that this parameter p could also be restricted to $\mathbb{N}$ for our purposes; we often use this below.

We say f is **absolutely approximable** (or $\mathcal{A}$-approximable) if some $\widetilde{f} \in \mathcal{A}_f$ is explicit. Likewise, f is **partially absolutely approximable** (or partially $\mathcal{A}$-approximable) if some $\widetilde{f} \in \mathcal{A}_f$ is partially explicit. Analogous definitions hold for f being **relatively approximable** (or $\mathcal{R}$-approximable) and **partially relatively approximable** (or partially $\mathcal{R}$-approximable). The concept of approximability (in the four variants here) is the starting point of our approach to real computation. Notice that "real approximation" amounts to "explicit computation on the base reals".

Remark on nominal domains of partial functions. It may appear redundant to consider a function f that is a partial function *and* whose nominal domain S is a proper subset of $\mathbb{R}^n$. In other words, by specifying $S = \mathbb{R}^n$ or $S = \texttt{domain}(f)$, we can either avoid partial functions, or avoid $S \neq \mathbb{R}^n$. This attitude is implicit in recursive function theory, for instance. It is clear that choice of S affects the computability of f since S determines the input to be fed to our computing devices. In the next section, the generic function $f(x) = \sqrt{x}$ is used to illustrate this fact. Intuitively, the definability of f at any point x is intrinsic to the function f,

[15] This is just a short hand for "$f : S \dashrightarrow \mathbb{R}$ and $S \subseteq \mathbb{R}^n$". Similarly, $f : S \subseteq \mathbb{R}^n \dashrightarrow T \subseteq \mathbb{R}$ is shorthand for $f : S \dashrightarrow T$ with the indicated containments for S and T.

but its points of undefinability is only incidental to f (an artifact of the choice of S). Unfortunately, this intuition can be wrong: the points of undefinability of f can tell us much about the global nature of f. To see this, consider the fact that the choice of S is less flexible in algebra than in analysis. In algebra, we are not free to turn the division operation in a field into a total function, by defining it only over non-zero elements. In analysis, it is common to choose S so that f behave nicely: e.g., f has no singularity, f is convergent under Newton iteration, etc. But even here, this choice is often not the best and may hide some essential difficulties. So in general, we do not have the option of specifying $S = \mathbb{R}^n$ or $S = \texttt{domain}(f)$ for a given problem.

Much of what we say in this and the next section are echos of themes found in [23, 39]. Our two main goals are (i) to develop the computability of f in the setting of a general nominal domain S, and (ii) to expose the connection between computability of f with its approximability. A practical theory of real computability in our view should be largely about approximability.

Regular Functions. Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. In [23] and [39], the real functions are usually restricted to $S = [a, b], (a, b)$ or $S = \mathbb{R}$; this choice is often is essential to the computability of f. To admit S which goes beyond the standard choices, we run into pathological examples such as $S = \mathbb{R} \setminus \mathbb{F}$. This example suggests that we need an ample supply of base reals in S. We say that a set $S \subseteq \mathbb{R}$ is **regular** if for all $x \in S$ and $n \in \mathbb{N}$, there exists $y \in S \cap \mathbb{F}$ such that $y = x \pm 2^{-n}$. Thus, S contains base reals arbitrarily close to any member. We say f is **regular** if $\texttt{domain}(f)$ is regular. Note that regularity, like all our approximability concepts, is defined relative to $\mathbb{F}$.

Cauchy Functions. The case $n = 0$ in (6) is rather special: in this case, f is regarded as a constant function, representing some real number $x \in \mathbb{R}$. An absolute approximation of x is any function $\widetilde{f} : \mathbb{F} \to \mathbb{F}$ where $\widetilde{f}(p) = x \pm 2^{-p}$ for all $p \in \mathbb{F}$. We call $\widetilde{f}$ a **Cauchy function** for x. The sequence $(\widetilde{f}(0), \widetilde{f}(1), \widetilde{f}(2), \ldots)$ is sometimes called a **rapidly converging Cauchy sequence** for x; relative to $\widetilde{f}$, the p**-th Cauchy convergent** of x is $\widetilde{f}(p)$.

Extending the above notation, we may write $\mathcal{A}_x$ for the set of all Cauchy functions for x. But note that $\widetilde{f}$ is not just an approximation of x, but it uniquely identifies x. Thus $\widetilde{f}$ is a representation of x. So by our underbar convention, we prefer to write "$\underline{x}$" for any Cauchy function of x. Also write "$\underline{x}[p]$" (instead of $\underline{x}(p)$) for the pth convergent of $\underline{x}$.

We can also let $\mathcal{R}_x$ denote the set of relative approximations of x. If some $\underline{x} \in \mathcal{A}_x$ ($\underline{x} \in \mathcal{R}_x$) is explicit, we say x is $\mathcal{A}$**-approximable** ($\mathcal{R}$**-approximable**). Below we show that x is $\mathcal{R}$-approximable iff x is $\mathcal{A}$-approximable. Hence we may simply speak of "approximable reals" without specifying whether we are concerned with absolute or relative errors.

Among the Cauchy functions in $\mathcal{A}_x$, we identify one with nice monotonicity properties: every real number x can be written as

$$n + 0.b_1 b_2 \cdots$$

where $n \in \mathbb{Z}$ and $b_i \in \{0, 1\}$. The b_i's are uniquely determined by x when $x \notin \mathbb{D}$. Otherwise, all b_i's are eventually 0 or eventually 1. For uniqueness, we require all b_i's to be eventually 0. Using this unique sequence, we define the **standard Cauchy function** of x via

$$\beta_x[p] = n + \sum_{i=1}^{p} b_i 2^{-i}.$$

For instance, $-5/3$ is written $-2 + 0.01010101 \cdots$. This defines the Cauchy function $\beta_x[p]$ for all $p \in \mathbb{N}$. Technically, we need to define $\beta_x[p]$ for all $p \in \mathbb{F}$: when $p < 0$, we simply let $\beta_x[p] = \beta_x[0]$; when $p > 0$ and is not an integer, we let $\beta_x[p] = \beta_x[\lceil p \rceil]$. We note some useful facts about this standard function:

LEMMA 13. *Let $x \in \mathbb{R}$ and $p \in \mathbb{N}$.*
(i) $\beta_x[p] \leq \beta_x[p+1] \leq x$.
(ii) $x - \beta_x[p] < 2^{-p}$.
(iii) If $y = \beta_x[p] \pm 2^{-p}$, then for all $n \leq p$, we also have $y = \beta_x[n] \pm 2^{-n}$. In particular, there exists $\underline{y} \in \mathcal{A}_y$ such $\underline{y}[n] = \beta_x[n]$ for all $n \leq p$.

(iv) There is a recursive procedure $B : \mathbb{F} \times \mathbb{N} \to \mathbb{F}$ such that for all $x \in \mathbb{F}, p \in \mathbb{N}$, $B(x,p) = \beta_x[p]$. In particular, for each $x \in \mathbb{F}$, the standard Cauchy function of x is recursive.

To see (iii), it is sufficient to verify that if $y = \beta_x[p] \pm 2^{-p}$ then $y = \beta_x[p-1] \pm 2^{1-p}$. Now $\beta_x[p] = \beta_x[p-1] + \delta 2^{-p}$ where $\delta = 0$ or 1. Hence $y = \beta_x[p] \pm 2^{-p} = (\beta_x[p-1] \pm 2^{-p}) \pm 2^{-p} = \beta_x[p-1] \pm 2^{1-p}$.

Explicit computation with one real transcendental. In general, it is not known how to carry out explicit computations (e.g., decide zero) in transcendental extensions of $\mathbb{Q}$ (but see [12] for a recent positive result). However, consider the field $F(\alpha)$ where α is transcendental over F. If F is ordered, then the field $F(\alpha)$ can also be ordered using an ordering where $a <' \alpha$ for all $a \in F$. Further, if F is explicit, then $F(\alpha)$ is also an explicit ordered field with this ordering $<'$. But the ordering $<'$ is clearly non-Archimedean (i.e., there are elements $a, x \in F(\alpha)$ such that for all $n \in \mathbb{N}$, $n|a| < |x|$). Now suppose $F \subseteq \mathbb{R}$ and $\alpha \in \mathbb{R}$ (for instance, $F = \mathbb{Q}$ and $\alpha = \pi$). Then $F(\alpha) \subseteq \mathbb{R}$ can be given the standard (Archimedean) ordering $<$ of the reals.

THEOREM 14. *If $F \subseteq \mathbb{R}$ is an explicit ordered field, and $\alpha \in \mathbb{R}$ is an approximable real that is transcendental over F, then the field $F(\alpha)$ with the Archimedean order $<$ is an explicit ordered field.*

Proof. The field $F(\alpha)$ is isomorphic to the quotient field of $F[X]$, and by Lemma 7(iii,iv), this field is explicit. It remains to show that the Archimedean order $<$ is explicit. Let $P(\alpha)/Q(\alpha) \in F(\alpha)$ where $P(X), Q(X) \in F[X]$ and $Q(X) \neq 0$. It is enough to show that we can recognize the set of positive elements of $F(\alpha)$. Now $P(\alpha)/Q(\alpha) > 0$ iff $P(\alpha)Q(\alpha) > 0$. So it is enough to recognize whether $P(\alpha) > 0$ for any $P(\alpha) \in F[\alpha]$. First, we can verify that $P(\alpha) \neq 0$ (this is true iff some coefficient of $P(\alpha)$ is nonzero). Next, since α is approximable, we find increasingly better approximations $\underline{\alpha}[p] \in \mathbb{F}$ of α, and evaluate $P(\underline{\alpha}[p])$ for $p = 0, 1, 2, \ldots$. To estimate the error, we derive from Taylor's expansion the bound $P(\alpha) = P(\underline{\alpha}[p]) \pm \delta_p$ where $\delta_p = \sum_{i \geq 1} 2^{-ip} |P^{(i)}(\underline{\alpha}[p])|$. We can easily compute an upper bound $\beta_p \geq |\delta_p|$, and stop when $|P(\underline{\alpha}[p])| > \beta_p$. Since $\delta_p \to 0$ as $p \to \infty$, we can also ensure that $\beta_p \to 0$. Hence termination is assured. Upon termination, we know that $P(\alpha)$ has the sign of $P(\underline{\alpha}[p])$. **Q.E.D.**

In particular, this implies that $\mathbb{D}(\pi)$ or $\mathbb{Q}(e)$ can serve as the set $\mathbb{F}$ of base reals. The choice $\mathbb{D}(\pi)$ may be appropriate in computations involving trigonometric functions, as it allows exact representation of the zeros of such functions, and thus the possibility to investigate the neighborhoods of such zeros computationally. Moreover, we can extend the above technique to any number of transcendentals, provided they are algebraically independent. For instance, π and $\Gamma(1/3) = 2.678938\ldots$ are algebraically independent and so $\mathbb{D}(\pi, \Gamma(1/3))$ would be an explicit ordered field.

Real predicates. Given $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$, define the predicate $\mathrm{Sign}_f : S \dashrightarrow \{-1, 0, 1\}$ given by

$$\mathrm{Sign}_f(x) = \begin{cases} 0 & \text{if } f(x) = 0, \\ +1 & \text{if } f(x) > 0, \\ -1 & \text{if } f(x) < 0, \\ \uparrow & \text{else.} \end{cases}$$

Define the related predicate $\mathrm{Zero}_f : S \dashrightarrow \{0, 1\}$ where $\mathrm{Zero}_f(x) \equiv |\mathrm{Sign}_f(x)|$ (so $\mathtt{range}(\mathrm{Zero}_f) \subseteq \{0, 1\}$). By the fundamental analysis of EGC (see Introduction), Sign_f is the critical predicate for geometric algorithms. We usually prefer to focus on the simpler Zero_f predicate because the approximability of these two predicates are easily seen to be equivalent in our setting of base reals (cf. [41]).

In general, a **real predicate** is a function $P : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$ where $\mathtt{range}(P)$ is a finite set. The approximation of real predicates is somewhat simpler than that of general real functions.

To treat the next result, we need some new definitions. Let $S \subseteq D \subseteq \Sigma^*$. We say S is **recursive modulo** D if there is a Turing machine that, on input x taken from the set D, halts in the state $q_\downarrow$ if $x \in S$, and in the state $q_\uparrow$ if $x \notin S$. Similarly, S is **partial recursive modulo** D if there is a Turing machine that, on input x taken from D, halts iff $x \in S$. Let $S \subseteq D \subseteq U$ where U is a ν-explicit set and $\nu : \Sigma^* \dashrightarrow U$. We say S is a (partially) **ν-explicit subset of U modulo** D if the set $\{w \in \Sigma^* : \nu(w) \in S\}$ is (partial) recursive modulo $\{w \in \Sigma^* : \nu(w) \in D\}$. Also, denote by $\mathtt{range}_{\mathbb{F}}(f) := \{f(x) : x \in \mathbb{F} \cap S\}$, the range of f when its domain is restricted to base real inputs.

LEMMA 15. *For a real predicate $P : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$, the following are equivalent:*
(i) P is partially $\mathcal{R}$-approximable
(ii) P is partially $\mathcal{A}$-approximable
(iii) Each $a \in \mathtt{range}_{\mathbb{F}}(P)$ is a computable real number and the set $P^{-1}(a) \cap \mathbb{F}$ is partially explicit modulo S.

Proof. (i) implies (ii): Let $\widehat{P} \in \mathcal{R}_P$ be a partially explicit function. Our goal is to compute some $\widetilde{P} \in \mathcal{A}_P$. Let the input for $\widetilde{P}$ be $(x, p) \in S \times \mathbb{N}$. First, compute $y = \widehat{P}(x)[1]$. If $y = \uparrow$, then we do not halt. So let $y = \downarrow$. We know that $P(x) = 0$ iff $\widehat{P}(x)[1] = 0$ ([41]). Hence we can define $\widetilde{P}(x)[p] = 0$ when $y = 0$. Now assume $y \neq 0$. Then $|y| \geq |P(x)|/2$, and we may compute and output $z := \widehat{P}(x)[p + 1 + \lceil \log_2 |y| \rceil]$. This output is correct since $z = P(x)(1 \pm 2^{-p-1-\lceil \log_2 |y| \rceil}) = P(x) \pm 2^{-p}$.

(ii) implies (iii): Let $\widetilde{P} \in \mathcal{A}_P$ be a partially explicit function. Fix any $a \in \mathtt{range}_{\mathbb{F}}(P)$. To see that a is a computable real, for all p, we can compute $\underline{a}[p]$ as $\widetilde{P}(x)[p]$ (for some fixed $x \in P^{-1}(a) \cap \mathbb{F}$). To show that $P^{-1}(a) \cap \mathbb{F}$ is a partially explicit subset of $\mathbb{F}$ modulo S, note that there is a p' such that for all $b, b' \in \mathtt{range}_{\mathbb{F}}(P)$, $b \neq b'$ implies $|b - b'| \geq 2^{-p'}$. We then choose an $\widetilde{a} \in \mathbb{F}$ such that $\widetilde{a}$ is a $(p' + 3)$-bit absolute approximation of a. Then we verify that $P^{-1}(a) \cap \mathbb{F}$ is equal to

$$\{x \in S : |\widetilde{P}(x)[p' + 3] - \widetilde{a}| \leq 2^{-p'-1}\}.$$

Thus, given $x \in S$, we can partially decide if $x \in P^{-1}(a)$ by first computing $\widetilde{P}(x)[p' + 3]$. If this computation halts, then $x \in P^{-1}(a)$ iff $|\widetilde{a} - \widetilde{P}(x)[p' + 3]| \leq 2^{-p'-1}$. Thus $P^{-1}(a) \cap \mathbb{F}$ is a partially explicit subset of $\mathbb{F}$ modulo S.

(iii) implies (i): Given $x \in S$ and $p \in \mathbb{N}$, we want to compute some $z = P(x)(1 \pm 2^{-p})$. We first determine the $a \in \mathtt{range}_{\mathbb{F}}(P)$ such that $P(x) = a$. We can effectively find a by enumerating the elements of $P^{-1}(b) \cap \mathbb{F} \cap S$ for each $b \in \mathtt{range}_{\mathbb{F}}(P)$ until x appears (this process does not halt iff $P(x) = \uparrow$). Assume a is found. If $a = 0$, then we simply output 0. Otherwise, we compute $\underline{a}[i]$ for $i = 0, 1, 2, \ldots$ until $|\underline{a}[i]| > 2^{-i}$. Let i_0 be the index when this happens. Then we have $|a| > b := |\underline{a}[i_0]| - 2^{-i_0}$. Set $q := p - \lfloor \log_2 b \rfloor$ and output $z := \underline{a}[q]$. As for correctness, note that $z = a \pm 2^{-q} = a \pm 2^{-p} b = a(1 \pm 2^{-p})$. **Q.E.D.**

There is an analogous result where we remove the "partially" qualifications in the statement of this lemma. However in (iii), we need to add the requirement that the set $S \setminus \mathtt{domain}(f)$ must be explicit relative to S. In view of this lemma, we can simply say that a real predicate is "(partially) approximable" instead of (partially) $\mathcal{A}$-approximable or $\mathcal{R}$-approximable. This lemma could be extended to "generalized predicates" $P : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$ whose range is **discrete** in the sense that for some $\varepsilon > 0$, for all $x, y \in \mathbb{R}^n$, $P(x) = P(y) \pm \varepsilon$ implies $P(x) = P(y)$.

On relative versus absolute approximability. In [41], we proved that a partial function $f : \mathbb{R} \dashrightarrow \mathbb{R}$ is $\mathcal{R}$-approximable iff it is $\mathcal{A}$-approximable and Zero_f is explicit. The proof extends to:

THEOREM 16. *Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. Then f is (partially) $\mathcal{R}$-approximable iff f is (partially) $\mathcal{A}$-approximable, and Zero_f is (partially) approximable.*

Relative approximation is dominant in numerical analysis: machine floating systems are all based on relative precision (for example, the IEEE Standard). See Demmel et al [15, 14] for recent work in this connection. Yet the analytic school exclusively discusses absolute approximations. This theorem shows why: relative approximation requires solving the zero problem, which is undecidable in the analytic approach.

5 Computable Real Functions

We now study computable real functions following the analytic school. Our main goal is to show the exact relationship between the approximation approach and the analytic school.

Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ be a partial real function. Following Ko [23], we will use **oracle Turing machines** (OTM) as devices for computing f. A real input x is represented by any Cauchy function $\underline{x} \in \mathcal{A}_x$. An OTM M has, in addition to the usual tape(s) of a standard Turing machine, two special tapes, called the **oracle tape** and the **precision tape**. It also has two special states,

$$q_?, q_! \tag{7}$$

called the **query state** and the **answer state**. We view M as computing a function (still denoted) $M : \mathbb{R} \times \mathbb{F} \to \mathbb{R}$. A real input x is represented by an arbitrary $\underline{x} \in \mathcal{A}_x$, which serves as an oracle. The input $p \in \mathbb{F}$ is placed on the precision tape. Whenever the computation of M enters the query state $q_?$, we require the oracle tape to contain a binary number $\underline{k}$. In the next instant, M will enter the answer state $q_!$, and simultaneously the string $\underline{k}$ is replaced by a representation of the kth convergent $\underline{x}[k]$. Then M continues computing, using this oracle answer. Eventually, there are two possible outcomes: either M loops and we write $M(\underline{x}, p) = \uparrow$, or it halts and its output tape holds a representation $\underline{d}$ for some $d \in \mathbb{F}$, and this defines the output, $M(\underline{x}, p) = d$. Equivalently, we write[16] "$M^{\underline{x}}[p] = d$". We say M **computes** a function $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$, if for all $x \in S$ and $p \in \mathbb{F}$, we have that for all $\underline{x} \in \mathcal{A}_x$,

$$M^{\underline{x}}[p] = \begin{cases} \uparrow & \text{if } f(x) = \uparrow \\ f(x) \pm 2^{-p} & \text{if } f(x) = \downarrow . \end{cases}$$

This definition extends to computation of multivariate partial functions of the form $f : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$. Then the OTM M takes as input n oracles (corresponding to the n real arguments). Each query on the oracle tape (when we enter state $q_?$) consists of a pair $(\underline{i}, \underline{k})$ indicating that we want the kth convergent of the ith oracle. Since such an extension will be trivial to do in most of our results, we will generally stay with the univariate case.

Our main definition is this: a real function $f : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$ is **computable** if there is a OTM that computes f. When f is total on S, we also say that it is **total computable**. In case $n = 0$, f denotes a real number x; the corresponding OTM is actually an ordinary Turing machine since we never use the oracle tape. In the analytic approach, x is known as a **computable real**. But it is easy to see that it is equivalent to x being an approximable real.

Example: consider $f(x) = \sqrt{x}$ where $f(x) = \downarrow$ iff $x \geq 0$. If $S = \mathbb{R}$, then it is easy to show that f is not computable by oracle Turing machines (the singularity at $x = 0$ cannot be decided in finite time). If we choose $S = [0, \infty)$, then f can be shown to be computable by oracle Turing machines even though the singularity at $x = 0$ remains. But we might prefer $S = [1, 4)$ in implementations since $\sqrt{x}$ can be reduced to $\sqrt{y}$ where $y \in [1, 2)$, by first computing $y = x \cdot 4^n$, $n \in \mathbb{Z}$.

Notice that our definition of "computable functions" include[17] partial functions. In this case, improper inputs for OTM's are handled in the conventional way of computability theory, with the machine looping. This follows Ko [23, p. 62]. We do not require OTM's to be halting because it would result in there being only trivial examples of computable real functions that are not total.

LEMMA 17. *If $f : S \subseteq \mathbb{R}^n \dashrightarrow \mathbb{R}$ is computable then f is partially $\mathcal{A}$-approximable.*

Proof. There is nothing to show if $n = 0$. To simplify notations, assume $n = 1$ (the case $n > 1$ is similar). Let M be an OTM that computes f. We define a function $\widetilde{f} \in \mathcal{A}_f$ where $\widetilde{f} : \mathbb{F}^2 \dashrightarrow \mathbb{F}$. To compute $\widetilde{f}$, we use an ordinary Turing machine N that, on input $d, p \in \mathbb{F}$, simulates M on the oracle β_d (the standard Cauchy function of d) and p. Note that by Lemma 13(iv), we can compute $\beta_d[k]$ for any $k \in \mathbb{N}$. If $M^{\beta_d}[p]$ outputs z, N will output z. Clearly, the choice $\widetilde{f}(d)[p] = z$ is correct. If $d \notin \mathtt{domain}(f)$, the computation loops. **Q.E.D.**

Recursively open sets. Let $\phi : \mathbb{N} \to \mathbb{F}$. This function defines a sequence $(I_0, I_1, I_2, \ldots)$ of open intervals where $I_n = (\phi(2n), \phi(2n+1))$. We say ϕ is an **interval representation** of the open set $S = \bigcup_{n \geq 0} I_n$. A set $S \subseteq \mathbb{R}$ is **recursively open** if S has an interval representation ϕ that is explicit. We say S is **recursively closed** if its complement $\mathbb{R} \setminus S$ is recursively open.

Note that if $\phi(2n) \geq \phi(2n+1)$ then the open interval $(\phi(2n), \phi(2n+1))$ is empty. In particular, the empty set $S = \emptyset$ is recursively open. So is $S = \mathbb{R}$.

The following result is from Ko [23, Theorem 2.31].

PROPOSITION 18. *A set $T \subseteq \mathbb{R}$ is recursively open iff there is a computable function $f : \mathbb{R} \dashrightarrow \mathbb{R}$ with $T = \mathtt{domain}(f)$.*

[16] This follows the convention of putting the oracle argument in the superscript position, and our convention of putting the precision parameter in square brackets.

[17] So alternatively, we could say that "f is *partial* computable" instead of "f is computable".

Modulus of continuity. Consider a partial function $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. We say f is **continuous** if for all $x \in \mathtt{domain}(f)$ and $\delta > 0$, there is an $\epsilon = \epsilon(x, \delta) > 0$ such that if $y \in \mathtt{domain}(f)$ and $y = x \pm \epsilon$ then $f(y) = f(x) \pm \delta$. We say f is **uniformly continuous** if for all $\delta > 0$, there is an $\epsilon = \epsilon(\delta) > 0$ such that for all $x, y \in \mathtt{domain}(f)$, $y = x \pm \epsilon$ implies $f(y) = f(x) \pm \delta$.

Let $m : S \times \mathbb{N} \dashrightarrow \mathbb{N}$. We call m a **modulus function** if for all $x \in S, p \in \mathbb{N}$, we have $m(x, p) = \uparrow$ iff $m(x, 0) = \uparrow$. We define $\mathtt{domain}(m) := \{x \in S : m(x, 0) = \downarrow\}$. Such a function m is called a **modulus of continuity** (or simply, modulus function) for f if $\mathtt{domain}(f) = \mathtt{domain}(m)$ and for all $x, y \in \mathtt{domain}(f), p \in \mathbb{N}$, if $y = x \pm 2^{-m(x,p)}$ then $f(y) = f(x) \pm 2^{-p}$.

Call a function $m : \mathbb{N} \to \mathbb{N}$ a **uniform modulus of continuity** (or simply, a uniform modulus function) **for** f if for all $x, y \in \mathtt{domain}(f), p \in \mathbb{N}$, if $y = x \pm 2^{-m(p)}$ then $f(y) = f(x) \pm 2^{-p}$. To emphasize the distinction between uniform modulus function and the non-uniform version, we might describe the latter as "local modulus functions". The following is immediate:

LEMMA 19. *Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. Then f is continuous iff it has a modulus of continuity. Then f is uniformly continuous iff it has a uniform modulus of continuity.*

Ko [23] uses uniform[18] continuity to characterize computable total real functions of the form $f : [a, b] \to \mathbb{R}$. Our goal is to generalize his characterization to capture real functions with non-compact domains, as well as those that are partial. Our results will characterize computable real functions $f : S \dashrightarrow \mathbb{R}$ where $S \subseteq \mathbb{R}^n$ is regular. Use of local continuity simplifies proofs, and avoids any appeal to the Heine-Borel theorem.

Multivalued Modulus function of an OTM. We now show how modulus functions can be computed, but they must be generalized to multivalued functions. Such functions are treated in Weihrauch [39]; in particular, computable modulus functions are multivalued [39, Cor. 6.2.8]. Let us begin with the usual (single-valued) modulus function $m : S \times \mathbb{N} \dashrightarrow \mathbb{N}$. It is computed by OTMs since one of m's arguments is a real number. If $(\underline{x}, p)$ is the input to an OTM N which computes m, we still write "$N^{\underline{x}}[p]$" for the computation of N on $(\underline{x}, p)$. Notice that, since the output of N comes from the discrete set $\mathbb{N}$, we do not need an extra "precision parameter" to specify the precision of the output (unlike the general situation when the output is a real number).

Let M be an OTM that computes some function $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. Consider the Cauchy function $\beta_x^+ \in \mathcal{A}_x$ where $\beta_x^+[n] = \beta_x[n + 1]$ for all $n \in \mathbb{N}$. (Thus β_x^+ is a "sped up" form of the standard Cauchy function β_x.) Consider the function $k_M : S \times \mathbb{N} \dashrightarrow \mathbb{N}$ where

$$k(x, p) = k_M(x, p) := \text{the largest } k \text{ such that the computation of } M^{\beta_x^+}[p] \text{ queries } \beta_x^+[k]. \tag{8}$$

This definition depends on M using β_x^+ as oracle. If $M^{\beta_x^+}[p] = \uparrow$ then $k(x, p) = \uparrow$; if $M^{\beta_x^+}[p] = \downarrow$ but the oracle was never queried, define $k(x, p) = 0$. Some variant of $k(x, p)$ was used by Ko to serve as a modulus function for f. Unfortunately, we do not know how to compute $k(x, p)$ as that seems to require simulating the oracle β_x^+ using an arbitrary oracle $\underline{x} \in \mathcal{A}_x$. Instead, we proceed as follows. For any $\underline{x} \in \mathcal{A}_x$, let $\underline{x}^+$ denote the oracle in $\mathcal{A}_x$ where $\underline{x}^+[n] = \underline{x}[n + 1]$ for all n. Define, in analogy to (8), the function $\underline{k}_M : \mathcal{A}_S \times \mathbb{N} \dashrightarrow \mathbb{N}$ where $\mathcal{A}_S = \cup\{\mathcal{A}_x : x \in S\}$ and

$$\underline{k}(\underline{x}, p) = \underline{k}_M(\underline{x}, p) := \text{the largest } k \text{ such that the computation of } M^{\underline{x}^+}[p] \text{ queries } \underline{x}^+[k]. \tag{9}$$

As before, if $M^{\underline{x}^+}[p] = \uparrow$ (resp., if the oracle was never queried), then $\underline{k}(\underline{x}, p) = \uparrow$ (resp., $\underline{k}(\underline{x}, p) = 0$). The first argument of $\underline{k}$ is an oracle, not a real number. For different oracles from the set $\mathcal{A}_x$, we might get different results; such functions $\underline{k}$ are[19] called an **intensional functions**. Computability of intensional functions are defined using OTM's, just as for real functions. We can naturally interpret $\underline{k}_M$ as representing a **multivalued function** (see [39, Section 1.4]) which may be denoted[20] $k_M : S \times \mathbb{N} \to 2^{\mathbb{N}}$ with $k_M(x, p) = \{\underline{k}_M(\underline{x}, p) : \underline{x} \in \mathcal{A}_x\}$. Statements about k_M can be suitably interpreted as statements about $\underline{k}_M$ (see below).

The following lemma is key:

[18] Indeed, uniform modulus functions are simply called "modulus functions" in Ko. He has a notion of "generalized modulus function" that is similar to our local modulus functions.

[19] Intensionality is viewed as the (possible) lack of "extensionality". We say $\underline{k}$ is **extensional** if for all $a, b \in \mathcal{A}_x$ and $p \in \mathbb{N}$, $\underline{k}(a, 0) = \downarrow$ iff $\underline{k}(a, p) = \downarrow$; moreover, $\underline{k}(a, p) \equiv \underline{k}(b, p)$. Extensional functions can be interpreted as (single-valued) partial functions on real arguments.

[20] Or, $k_M : S \times \mathbb{N} \rightrightarrows \mathbb{N}$.

LEMMA 20. *Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ be computed by an OTM M, and $p \in \mathbb{N}$.*
(i) If $x \in \mathtt{domain}(f)$ and $\underline{x} \in \mathcal{A}_x$, then $(x \pm 2^{-\underline{k}_M(\underline{x},p)}) \cap S \subseteq \mathtt{domain}(f)$.
(ii) If, in addition, $y = x \pm 2^{-\underline{k}_M(\underline{x},p)}$ and $y \in S$ then $f(y) = f(x) \pm 2^{1-p}$.

Proof. Let $y = x \pm 2^{-\underline{k}_M(\underline{x},p)}$ where $x \in \mathtt{domain}(f)$ and $y \in S$.
(i) We must show that $y \in \mathtt{domain}(f)$. For this, it suffices to show that M halts on the input $(\underline{y}, p)$ for some $\underline{y} \in \mathcal{A}_y$. Consider the modified Cauchy function $\underline{y}'$ given by

$$\underline{y}'[n] = \begin{cases} \underline{x}^+[n] & \text{if } n+1 \leq \underline{k}_M(\underline{x},p) \\ \underline{y}[n] & \text{else.} \end{cases}$$

To see that $\underline{y}' \in \mathcal{A}_y$, we only need to verify that $y = \underline{y}'[n] \pm 2^{-n}$ for $n \leq \underline{k}_M(\underline{x},p)$. This follows from

$$y = x \pm 2^{-\underline{k}_M(\underline{x},p)} = (\underline{x}[n+1] \pm 2^{-n-1}) \pm 2^{-\underline{k}_M(\underline{x},p)} = \underline{x}^+[n] \pm 2^{-n} = \underline{y}'[n] \pm 2^{-n}.$$

Since the computation of $M^{\underline{x}^+}[p]$ does not query $\underline{x}^+[n]$ for $n > \underline{k}_M(\underline{x},p)$, it follows that this computation is indistinguishable from the computation of $M^{\underline{y}'}[p]$. In particular, both $M^{\underline{x}^+}[p]$ and $M^{\underline{y}'}[p]$ halt. Thus $y \in \mathtt{domain}(f)$.
(ii) We further show

$$\begin{aligned} |f(y) - f(x)| &\leq |f(y) - M^{\underline{y}'}[p]| + |M^{\underline{x}^+}[p] - f(x)| \\ &\leq 2^{-p} + 2^{-p} = 2^{1-p}. \end{aligned}$$

Q.E.D.

Let M be an OTM. Define the intensional function $\underline{m} : \mathcal{A}_S \times \mathbb{N} \dashrightarrow \mathbb{N}$ by

$$\underline{m}(\underline{x}, p) := \underline{k}_M(\underline{x}, p+1). \tag{10}$$

We call $\underline{m}$ an (intensional) **modulus of continuity** for $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ if[21] for all $x \in S$ and $p \in \mathbb{N}$, we have $\underline{m}(\underline{x}, p) = \downarrow$ iff $x \in \mathtt{domain}(f)$. In addition, for all $x, y \in \mathtt{domain}(f)$, if $y = x \pm 2^{-\underline{m}(\underline{x},p)}$ then $f(y) = f(x) \pm 2^{-p}$. The multivalued function $m : S \times \mathbb{N} \to 2^{\mathbb{N}}$ corresponding to $\underline{m}$ will be called a **multivalued modulus of continuity** of f. In this case, define $\mathtt{domain}(m) = \mathtt{domain}(\underline{m})$ to be $\mathtt{domain}(f)$. It is easy to see that f has a multivalued modulus of continuity iff it has a (single-valued) modulus of continuity. The next result may be compared to [39, Corollary 6.2.8].

LEMMA 21. *If $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ is computed by an OTM M then the function $\underline{m}(\underline{x}, p)$ of (10) is a modulus of continuity for f. Moreover, $\underline{m}$ is computable.*

Proof. To show that $\underline{m}$ is a modulus of continuity for f, we may assume $x \in S$. Consider two cases: if $x \notin \mathtt{domain}(f)$ then $\underline{k}(\underline{x}, p+1)$ is undefined. Hence $\underline{m}(\underline{x}, p)$ is undefined as expected. So assume $x \in \mathtt{domain}(f)$. Suppose $y = x \pm 2^{-\underline{m}(\underline{x},p)} = x \pm 2^{-\underline{k}_M(\underline{x},p+1)}$ and $y \in S$. By the previous lemma, we know that $y \in \mathtt{domain}(f)$ and $f(y) = f(x) \pm 2^{-p}$. Thus $\underline{m}(\underline{x}, p)$ is a modulus of continuity for f.

To show that $\underline{m}$ is computable, we construct an OTM N which, on input $\underline{x} \in \mathcal{A}_x$ and $p \in \mathbb{F}$, simulates the computation of $M^{\underline{x}^+}[p+1]$. Whenever the machine M queries $\underline{x}[n]$ for some n, the machine N queries $\underline{x}^+[n] = \underline{x}[n+1]$ instead. When the simulation halts, N outputs the largest k such that $\underline{x}^+[k]$ was queried (or $k = 0$ if there were no oracle queries). **Q.E.D.**

The above proof shows that $m(x, p) := k_M(x, p+1)$ is a modulus of continuity in the following "strong" sense: a multivalued modulus function m for $f : S \dashrightarrow \mathbb{R}$ is said to be **strong** if $x \in \mathtt{domain}(f)$ implies $[x \pm 2^{-\underline{m}(\underline{x},p)}] \cap S \subseteq \mathtt{domain}(f)$. Note that if S is regular, then $\mathtt{domain}(m)$ is regular. Thus:

COROLLARY 22. *If $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ is computable, then it has a strong multivalued modulus function that is computable. In particular, f is continuous.*

[21] In discussing intensional functions, it is convenient to assume that whenever we introduce a quantified real variable x, we simultaneously introduce a corresponding universally-quantified Cauchy function variable $\underline{x} \in \mathcal{A}_x$. These two variables are connected by our under-bar convention. That is, "$(Qx \in S \subseteq \mathbb{R})$" should be translated "$(Qx \in S \subseteq \mathbb{R})(\forall \underline{x} \in \mathcal{A}_x)$" where $Q \in \{\forall, \exists\}$.

Modulus cover. An alternative formulation of strong modulus of continuity is this: let

$$\square\mathbb{F} := \{(a,b) : a < b, a, b \in \mathbb{F}\}$$

denote the set of open intervals over $\mathbb{F}$. A **modulus cover** refers to any subset $G \subseteq \square\mathbb{F} \times \mathbb{N}$. For simplicity, the typical element in G is written (a,b,p) instead of the more correct $((a,b),p)$. We call G a **modulus cover of continuity** (or simply, a modulus cover) for $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$ if the following two conditions hold:

(a) For each $p \in \mathbb{N}$ and $x \in \mathtt{domain}(f)$, there exists $(a,b,p) \in G$ with $x \in (a,b)$.

(b) For all $(a,b,p) \in G$, we have $(a,b) \cap S \subseteq \mathtt{domain}(f)$. Moreover, $x, y \in (a,b) \cap S$ implies $f(x) = f(y) \pm 2^{-p}$.

If the characteristic function $\chi_G : \square\mathbb{F} \times \mathbb{N} \dashrightarrow \{1\}$ of G is (resp., partially) explicit then we say G is (resp., partially) explicit.

The advantage of using G over a modulus function m is that we avoid multivalued functions, and the triples of G are parametrized by base reals. Thus we compute the characteristic function of G using ordinary Turing machines while m must be computed by OTMs. We next show that we could interchange the roles of G and m.

LEMMA 23. *For $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$, the following statements are equivalent:*
(i) f has a modulus cover G that is partially explicit.
(ii) f has a strong multivalued modulus function m that is computable.

Proof. (i) implies (ii): If G is available, we can define $\underline{m}(\underline{x}, p)$ via the following dovetailing process: let the input be a Cauchy function $\underline{x}$. For each $(a,b,p) \in \square\mathbb{F} \times \mathbb{N}$, we initiate a (dovetailed) computation to do three steps:
(1) Check that $(a,b,p) \in G$.
(2) Find the first $i = 0, 1, \ldots$ such that $[\underline{x}[i] \pm 2^{-i}] \subseteq (a,b)$.
(3) Output $k = -\lfloor \log_2 \min\{\underline{x}[i] - 2^{-i} - a, b - 2^{-i} - \underline{x}[i]\} \rfloor$.

Correctness of this procedure: since G is partially explicit, step (1) will halt if $(a,b,p) \in G$. Step (2) amounts to checking the predicate $a < x < b$. If $x \in \mathtt{domain}(f)$ then steps (1) and (2) will halt for some $(a,b,p) \in G$. The output k in step (3) has the property that if $y = x \pm 2^{-k}$ and $y \in S$ then $y \in \mathtt{domain}(f)$ and $f(y) = f(x) \pm 2^{-p}$. Thus $\underline{m}$ is a strong modulus of continuity of f, and our procedure shows $\underline{m}$ to be computable.

(ii) implies[22] (i): Suppose f has a modulus function $\underline{m}$ that is computed by the OTM M. A finite sequence $\sigma = (x_0, x_1, \ldots, x_k)$ is called a **Cauchy prefix** if there exists a Cauchy function $\underline{x}$ such that $\underline{x}[i] = x_i$ for $i = 0, \ldots, k$. We say $\underline{x}$ **extends** σ in this case. Call σ a **witness** for a triple $(a,b,p) \in \mathbb{F} \times \mathbb{F} \times \mathbb{N}$ provided the following conditions hold:
(4) $[a,b] \subseteq \bigcap_{i=0}^{k} [x_i \pm 2^{-i}]$.
(5) If $\underline{x}$ extends σ then the computation $M^{\underline{x}}[p]$ halts and does not query the oracle for $\underline{x}[n]$ for any $n > k$.
(6) If $M^{\underline{x}}[p]$ outputs $\ell = \underline{m}(\underline{x}, p)$, then we have $0 < b - a < 2^{-\ell}$.

Let G comprise all (a,b,p) that has a witness. The set G is partially explicit since, on input (a,b,p), we can dovetail through all sequences $\sigma = (x_0, \ldots, x_k)$, checking if σ is a witness for (a,b,p). This amounts to checking conditions (4)-(6). To see that G is a modulus cover for f, we first note that if $(a,b,p) \in G$ then $(a,b) \cap S \subseteq \mathtt{domain}(M) = \mathtt{domain}(f)$. Moreover, for all $p \in \mathbb{N}$ and $x \in \mathtt{domain}(f)$, we claim that there is some $(a,b) \in \square\mathbb{F}$ where $(a,b,p) \in G$ and $x \in (a,b)$. To see this, consider the computation of $M^{\beta_x}[p]$ where β_x is the standard Cauchy function of x. If the largest query made by this computation to the oracle β_x is k, then consider the sequence $\sigma = (\beta_x[0], \ldots, \beta_x[k])$. Note that x is in the interior of $[\beta_x[k] \pm 2^{-k}] = \bigcap_{i=0}^{k} [\beta_x[i] \pm 2^{-i}]$. If $M^{\beta_x}[p] = \ell$, then we can choose $(a,b) \subseteq (\beta_x[k] \pm 2^{-k})$ such that $b - a \leq 2^{-\ell}$ and $a < x < b$. Also for all $y, y' \in (a,b)$, we have $y' = y \pm 2^{\underline{m}(\underline{y},p)}$ and hence $|f(y') - f(y)| \leq 2^{-p}$. **Q.E.D.**

COROLLARY 24. *If the function $f : S \dashrightarrow \mathbb{R}$ is computable then it has a partially explicit modulus cover G. Moreover, if S is regular then f is regular.*

[22] This proof is kindly provided by V.Bosserhoff and another referee. My original argument required S to be regular.

Proof. By Corollary 22 and Lemma 23, we see that such a G exists. Let S be regular. To see that f is regular, note that $x \in \texttt{domain}(f)$ implies that for all $p \in \mathbb{N}$, we have $x \in (a,b)$ for some $(a,b,p) \in G$. So $(x \pm \varepsilon) \subseteq (a,b)$ for sufficiently small $\varepsilon > 0$. Regularity of S implies $(x \pm \varepsilon) \cap S \cap \mathbb{F}$ is non-empty. But $(a,b) \cap S \subseteq \texttt{domain}(f)$ implies $(x \pm \varepsilon) \cap \texttt{domain}(f) \cap \mathbb{F} = (x \pm \varepsilon) \cap S \cap \mathbb{F}$. The non-emptiness of $(x \pm \varepsilon) \cap \texttt{domain}(f) \cap \mathbb{F}$ proves that $\texttt{domain}(f)$ is regular. **Q.E.D.**

Main Result. We now characterize computability of real functions in terms of two explicitness concepts ($\mathcal{A}$-approximability and explicit modulus cover). In one direction, we also need a regularity condition.

THEOREM 25 (Characterization of computable functions). *Let $f : S \subseteq \mathbb{R} \dashrightarrow \mathbb{R}$. If f is computable then the following two conditions hold:*
(i) f is partially $\mathcal{A}$-approximable.
(ii) f has a partially explicit modulus cover.
Conversely, if S is regular then (i) and (ii) implies f is computable.

Proof. If f is computable, then conditions (i) and (ii) hold by Lemma 17 and Corollary 24 (resp.). Conversely, suppose (i) f is partially $\mathcal{A}$-approximable via a partially explicit $\widetilde{f} \in \mathcal{A}_f$, and (ii) f has a modulus cover $G \subseteq \square\mathbb{F} \times \mathbb{N}$ that is partially explicit. Consider the following OTM M to compute f: given a Cauchy function $\underline{x} \in \mathcal{A}_x$ and precision $p \in \mathbb{N}$:
STEP 1: Perform a dovetailed computation over all $(k,a,b) \in \mathbb{N} \times \square\mathbb{F}$. For each (k,a,b), check if $[\underline{x}[k] \pm 2^{-k}] \subseteq (a,b)$ and $(a,b,p+1) \in G$.
STEP 2: Suppose (k,a,b) passes the test in STEP 1. We compute

$$n = 1 - \log_2 \left(\min\{\underline{x}[k] - 2^{-k} - a, b - \underline{x}[k] - 2^{-k}\} \right).$$

STEP 3: Perform a dovetailed computation over all $c \in \mathbb{F} \cap [\underline{x}[k] \pm 2^{-k} \pm 2^{-n}]$: for each c, compute $\widetilde{f}(c, p+1)$. If any such computation halts, output its result.

First we show partial correctness: if M outputs $z = \widetilde{f}(c, p+1)$, then $z = f(c) \pm 2^{-p-1}$. Since $f(c) = f(x) \pm 2^{-p-1}$, we conclude that $z = f(x) \pm 2^{-p}$, as desired. Now we show conditional termination of M: for $x \in S$, we show that $x \in \texttt{domain}(f)$ iff M halts on $\underline{x}, p$. But $x \in \texttt{domain}(f)$ iff $x \in (a,b)$ for some $(a,b,p+1) \in G$. Then for k large enough, $[\underline{x}[k] \pm 2^{-k}] \subseteq (a,b)$, and so STEP 1 will halt. Conversely, if $x \notin \texttt{domain}(f)$ then STEP 1 does not halt. We finally show the halting of STEP 3, assuming $x \in \texttt{domain}(f)$. By the regularity of S, we conclude that there exists $c \in I \cap S \cap \mathbb{F}$ where $I = [x \pm 2^{-k} \pm 2^{-n}]$ is the interval used in STEP 3. By definition of modulus cover, $c \in (a,b) \cap S \subseteq \texttt{domain}(f)$. Hence STEP 3 will halt when such a c is found. **Q.E.D.**

This theorem is important because it tells us exactly what we are giving up when we abandon computability for absolute approximability: we give up precisely one thing, continuity. That is exactly the effect we want in EGC, since continuous functions are too restrictive in our applications. We obtain a stronger characterization of f in the important case where $S = [a,b]$ (essentially [23, Corollary 2.14] in Ko). Now we need to invoke the Heine-Borel theorem:

THEOREM 26. *Let a, b be computable reals. A total function $f : [a,b] \to \mathbb{R}$ is computable iff it is $\mathcal{A}$-approximable and it has an explicit uniform modulus function.*

Proof. By the characterization theorem, computability of f is equivalent to (i) the partial $\mathcal{A}$-approximability of f, and (ii) existence of a partially explicit modulus cover G. Since f is a total function, it is $\mathcal{A}$-approximable. It remains to show that the existence of G is equivalent to f having an explicit uniform modulus function $m : \mathbb{N} \to \mathbb{N}$.

One direction is easy: if $m : \mathbb{N} \to \mathbb{N}$ is an explicit uniform modulus function for f, then we can define G to comprise all (c,d,p) such that that $c, d \in \mathbb{F}$, $c < d < c+1$ and $p \in \mathbb{N}$ satisfies $d - c \leq 2^{-m(p)}$. Conversely, suppose G exists. From part(a) in the definition of modulus cover of f, for any $p \in \mathbb{N}$, the set $\{(c,d) : (c,d,p) \in G\}$ of open intervals is a cover for $[a,b]$. By Heine-Borel, there is a finite subcover $C = \{I_j : j = 0, \ldots, k\}$. Wlog, C is a minimal cover. Then the intervals have a uniquely ordering $I_0 < I_1 < \cdots < I_k$ induced by sorting their left (equivalently, right) endpoints. Let $J_i = I_{i-1} \cap I_i$ $(i = 1, \ldots, k)$. By minimality of C, we see that the J_i's are pairwise disjoint. Let $w(C) = \min\{w(J_i) : i = 1, \ldots, k\}$ where $w(I) = d - c$

is the width of an interval $I = (c, d)$. It follows that if $x, y \in [a, b]$ and $|x - y| < w(C)$ then $\{x, y\} \subseteq I_j$ for some $j = 0, \ldots, k$. This would imply $f(x) = f(y) \pm 2^{-p}$. Therefore if we define $m(p) := -\lfloor \log_2 w(C) \rfloor$ then m would be a uniform modulus function for f.

To show that m is explicit, we convert the preceding outline into an effective procedure. To compute $m(p)$, we first search for a cover as follows: we dovetail all the computations to search for triples (c, d, p) in G (for all $c, d \in \mathbb{F}$). We maintain a current minimal set C of intervals, initially C is empty. Inductively, assume $\bigcup C$ is equal to the union of all intervals found so far. For each triple $(c, d, p) \in G$, we discard (c, d) if $(c, d) \subseteq \bigcup C$. Otherwise, we add (c, d) to C, and remove any resulting redundancies in C. Next, we check if $[a, b] \subseteq \bigcup C$, and if so we can compute $m(p) := -\lfloor \log_2 w(C) \rfloor$, as described above. If not, we continue the search for more intervals (c, d). We note that checking if $[a, b] \subseteq \bigcup C$ is possible since a, b are[23] computable.

Q.E.D.

6 Unified Framework for Algebraic-Numeric Computation

We have seen that our real approximability approach is a modified form of the analytic approach. We now address the algebraic approach. The main proponent of this approach is Blum, Cucker, Shub and Smale [6]. For an articulate statement of their vision, we refer to their manifesto [5], reproduced in [6, Chap. 1]. Many researchers have tried to refine and extend the algebraic model, by considering the Boolean part of its complexity classes, by introducing non-unit cost measures, by injecting an error parameter into complexity functions, etc. These refinements can be viewed as attempting to recover some measure of representational complexity into the algebraic model. We do not propose to refine the algebraic approach. Instead, we believe a different role is reserved for the algebraic approach.

To motivate this role, consider the following highly simplified two-stage scheme of how computational scientists goes about solving a practical numerical problem (e.g., solving a PDE model or a numerical optimization problem).

STEP A: First, we determine the abstract algorithmic **Problem** P to be solved: in the simplest form, this amounts to specifying the input and output in mathematical terms. We then design an **Ideal Algorithm** A. This algorithm assumes certain real operations such as $+, -, \times, \exp()$, etc. Algorithm A may also use standard computational primitives for discrete computation such as found in programming languages, and abstract data types such as queues or heaps. We then show that Algorithm A solves the problem P in this ideal setting.

STEP B: We proceed to implement Algorithm A as a **Numerical Algorithm** B in some actual programming language. Algorithm B must now address concrete representation issues: concrete data structures that implement abstract data types, possible error in the input data, approximation of the operations $+, -, \times, \exp()$, etc. We also define the sense in which an implementation constitutes an acceptable approximation of algorithm A, for example, in the sense of backwards analysis, or in the EGC sense (below). Finally, we prove that Algorithm B satisfies this requirement.

What is the conceptual view of these two steps? Basically, we are asking for computational models for Algorithms A and B. It seems evident that Algorithm A is a program in some algebraic model like the BSS model [6] or the Real RAM. Since Algorithm A uses the primitives $+, -, \times, \exp(), \ldots$, we say that its **computational basis** is the set $\Omega = \{+, -, \times, \exp(), \ldots\}$. What about Algorithm B? If we accept the Church-Turing Thesis, then we could say that Algorithm B belongs to the Turing model. This suggestion is appropriate if we are only interested in computability issues. But for finer complexity distinctions, we want a computational model that better reflects how numerical analysts design algorithms, a "numerical computational model" that is more structured than Turing machines. We outline such a model below.

Suppose we now have two computational models: an algebraic model α for Algorithm A, and a numerical model β for Algorithm B. We ask a critical question: can Algorithm A be implemented by some Program B? In other words, we want "Transfer Theorems" that assure us that every program of α can be successfully implemented as a program in β.

There is much (psychological) validity in this 2-stage scheme: certainly, theoretical computer scientists and computational geometers design algorithms this way, using the Real RAM model. But even numerical

[23] A referee pointed out that it suffices that a be a left-computable, and b a right-computable, real number.

analysts proceed in this manner. Indeed, most numerical analysis books take STEP A only, and rarely discuss the issues in taking STEP B. From a purely practical viewpoint, the algebraic model provides a useful level of abstraction that guide the eventual transfer of algorithmic ideas into actual executable programs. We thus see that on the one hand, the algebraic model is widely used, and on the other hand, severe criticisms arise when it is proposed as the computational model of numerical analysis (and indeed of all scientific computation). This tension is resolved through our scheme where the algebraic model takes its proper place.

Pointer Machines. We now face the problem of constructing a computational framework in which the algebraic and numerical worlds can co-exist and complement each other. We wish to ensure from the outset that both discrete combinatorial computation and continuous numerical computation can be naturally expressed in this framework. Following Knuth, we may describe such computation as "semi-numerical". Current theories of computation (algebraic or analytic or standard complexity theories) do not adequately address such problems. For instance, real computation is usually studied as the problem of computing a real function $f : \mathbb{R} \dashrightarrow \mathbb{R}$ even though this is a very special case with no elements of combinatorial computing. In Sections 4 and 5 we followed this tradition. On the other hand, algorithms of computational geometry are invariably semi-numerical [42]. Following [41], we will extend Schönhage's elegant Pointer Machine Model [36] to provide such a framework.

We briefly recall the concept of a pointer machine (or storage modification machine). Let Δ be any set of symbols, which we call **tags**. Pointer machines manipulates graphs whose edges are labeled by tags. More precisely, a **tagged graph** (or **Δ-graph**) is a finite directed graph $G = (V, E)$ with a distinguished node $s \in V$ called the **origin** and a label function that assigns tags to edges such that outgoing edges from any node have distinct tags. We can concisely write $G = (V, s, \tau)$ where $s \in V$ and $\tau : V \times \Delta \dashrightarrow V$ is the **tag function**. Note that τ is a partial function, and it implicitly defines E, the edge set: $(u, v) \in E$ iff $\tau(u, a) = v$ for some $a \in \Delta$. Write $u \xrightarrow{a} v$ if $\tau(u, a) = v$. The edge (u, v) is also called a **pointer**, written $u \to v$, and its tag is a. Each word $w \in \Delta^*$ defines at most one path in G that starts at the origin and follows a sequence of edges whose tags appear in the order specified by w. Let $[w]_G$ (or $[w]$ if G is understood) denote the last node in this path. If no such path exists, then $[w] = \uparrow$. It is also useful to let w^- denote the word where the last tag a in w is removed: so $w^- a = w$. In case $w = \epsilon$ (empty word), then let w^- denote ϵ. Thus, if $w \neq \epsilon$ then the last edge in the path is $[w^-] \to [w]$. A node is **accessible** if there is a w such that $[w] = u$; otherwise, it is inaccessible. If we prune all inaccessible nodes and edges issuing from them, we get a **reduced tagged graph**. We distinguish tagged graphs only up to **equivalence**, defined as isomorphism on reduced tagged graphs. For any Δ-graph G, let $G|w$ denote the Δ-graph that is identical to G except that the origin of G is replaced by $[w]_G$.

Let $\mathcal{G}_\Delta$ denote the set of all Δ-graphs. Pointer machines manipulate Δ-graphs. Thus, the Δ-graphs play the role of strings in Turing machines, and $\mathcal{G}_\Delta$ is the analogue of Σ^* as the universal representation set. The key operation[24] of pointer machines is the **pointer assignment instruction**: if $w, w' \in \Delta^*$, then the assignment

$$w \leftarrow w'$$

modifies the current Δ-graph G by redirecting or creating a single pointer. This operation is defined iff $[w^-]$ and $[w']$ are both defined. We have two possibilities: if $w = \epsilon$, then this amounts to changing the origin to $[w']$. Else, if $w = w^- a$ then the pointer from $[w^-]$ with tag a will be redirected to point to $[w']$. If there was no previous pointer with tag a, then this operation creates the new pointer $[w^-] \xrightarrow{a} [w']$. If G' denotes the Δ-graph after the assignment, we generally have the effect that $[w]_{G'} = [w']_G$. But this equation fails in general: e.g., let $w = abaa$ and $w' = ab$ where $[\epsilon], [a], [ab]$ are three distinct nodes but $[aba] = [\epsilon] = [abb]$. Assignment, plus three other instructions of the pointer machines, are summarized in rows (i)-(iv) of Table 1. A **pointer machine**, then, is a finite sequence of these four types of pointer instructions, possibly with labels. With suitable conventions for input, output, halting in state $q_\uparrow$ or $q_\downarrow$, etc, which the reader may readily supply (see [41] for details), we now see that each pointer machine computes a partial function $f : \mathcal{G}_\Delta \dashrightarrow \mathcal{G}_\Delta$. It is easy to see that pointer machines can simulate Turing machines. The converse simulation is also possible. The merit of pointer machines lies in their naturalness in modeling combinatorial computing – in particular, it can directly represent graphs, in contrast to Turing machines that must "linearize" graphs into strings.

[24] The description here is a generalization of the one in [41], which also made the egregious error of describing the result of $w \leftarrow w'$ by the equation $[w]_{G'} = [w']_G$.

Type	Name	Instruction	Effect (G is transformed to G')
(i)	Pointer Assignment	$w \leftarrow w'$	$[w^-]_G \xrightarrow{a} [w']_G$ holds in G' where $w^- a = w'$
(ii)	Node Creation	$w \leftarrow$ **new**	$[w^-]_G \xrightarrow{a} u$ holds in G' where u is a new node
(iii)	Node Comparison	**if** $w \equiv w'$ **goto** L	$G' = G$
(iv)	Halt and Output	**HALT**(w)	Output $G' = G\|w$
(v)	Value Comparison	**if** $(w \diamond w')$ **goto** L where $\diamond \in \{=, <, \leq\}$	Branch if $Val_G(w) \diamond Val_G(w')$
(vi)	Value Assignment	$w := o(w_1, \ldots, w_m)$ where $o \in \Omega$ and $w, w_i \in \Delta^*$	$Val_{G'}(w) = o(Val_G(w_1), \ldots, Val_G(w_n))$

Table 1: Instruction Set of Pointer Models

Semi-numerical Problems and Real Pointer Machines. Real RAM's and BSS-machines have the advantage of being natural for numerical and algebraic computation. We propose to marry these features with the combinatorial elegance of pointer machines.

We extend tagged graphs to support real computation by associating a real $\texttt{val}(u) \in \mathbb{R}$ with each node $u \in V$. Thus a **real Δ-graph** is given by $G = (V, s, \tau, \texttt{val})$. Let $\mathcal{G}_\Delta(\mathbb{R})$ (or simply $\mathcal{G}(\mathbb{R})$) denote the set of such **real Δ-graphs**. For a **real pointer machine** to manipulate such graphs, we augment the instruction set with two instructions, as specified by rows (v)-(vi) in Table 1. Instruction (v) compares the values of two nodes and branches accordingly; instruction (vi) applies an algebraic operation g to the values specified by nodes. The set of algebraic operations g comes from a set Ω of algebraic operations which we call the **computational basis** of the model. The simplest computational basis is $\Omega_0 = \{+, -, \times\} \cup \mathbb{Z}$, resulting in nodes with only integer values. Each real pointer machine computes a partial function

$$f : \mathcal{G}(\mathbb{R}) \dashrightarrow \mathcal{G}(\mathbb{R}) \tag{11}$$

For simplicity, we define[25] a **semi-numerical problem** to also be a partial function of the form (11). The objects of computational geometry can be represented by real tagged graphs (see [42]). Thus, the problems of computational geometry can be regarded as semi-numerical problems. A semi-numerical problem (11) is **Ω-solvable** if there is a halting real pointer machine over the basis Ω that solves it. Another example of a semi-numerical problem is the **evaluation function** $\text{Eval}_\Omega : Expr(\Omega) \dashrightarrow \mathbb{R}$ (cf. (4)) where the set $Expr(\Omega)$ of expressions is directly represented by tagged graphs.

Real pointer machines constitute our idealized algebraic model for STEP A in our 2-stage scheme. Since real pointer machines are equivalent in power to real RAMs or BSS machines, the true merit of real pointer machines lies in their naturalness for capturing semi-numerical problems. For STEP B, the Turing model is adequate[26] but not natural. For instance, numerical analysts do not think of their algorithms as pushing bits on a tape, but as manipulating higher-level objects such as numbers or matrices with appropriate representations.

To provide a model closer to this view, we introduce **numerical Δ-graphs** which is similar to real Δ-graphs except that the value at each node is a base real from $\mathbb{F}$. The instructions for modifying numerical tagged graphs are specified by rows (i)-(v) in Table 1, plus a modified row (vi). The modification is that each $g \in \Omega$ is replaced by a relative approximation $\widetilde{g}$ which takes an extra precision argument (a value in $\mathbb{F}$). So a **numerical pointer machine** N is defined by a sequence of these instructions; we assume a fixed convention for specifying a precision parameter p for such machines. N computes a partial function $\widetilde{f} : \mathcal{G}(\mathbb{F}) \times \mathbb{F} \dashrightarrow \mathcal{G}(\mathbb{F})$. Let $X = \mathcal{A}$ or $\mathcal{R}$. We say that $\widetilde{f}$ is an **X-approximation** of f if, for all $G \in \mathcal{G}(\mathbb{F})$ and $p \in \mathbb{F}$, the graph $\widetilde{f}(G, p)$ (if defined) is a p-bit X-approximation of $f(G)$ in this sense: their underlying reduced graphs are isomorphic, and each numerical value in $\widetilde{f}(G, p)$ is a p-bit X-approximation of the corresponding real value in $f(G)$. We say f is X-approximable if there is a halting numerical machine that computes an X-approximation of f. *This is the EGC notion of approximation.*

[25] That is analogous to defining problems in discrete complexity to be a function $f : \Sigma^* \dashrightarrow \Sigma^*$. So we side-step the issues of representation.

[26] We might also say that recursive functions are an adequate basis for semi-numerical problems. But it is even less natural.

Transfer Theorems. Let $\mathcal{SN}_\Omega$ denote the class of semi-numerical problems that are Ω-solvable by real pointer machines. For instance, $\text{Eval}_\Omega \in \mathcal{SN}_\Omega$. Similarly, $\widetilde{\mathcal{SN}}_\Omega$ is the class of semi-numerical problems that can be $\mathcal{R}$-approximated by numerical pointer machines. (Note that we use the relative approximation here.) What is the relationship between these two classes? We reformulate a basic result from [41, Theorem 23]:

PROPOSITION 27. *Let Ω be any set of real operators. Then $\mathcal{SN}_\Omega \subseteq \widetilde{\mathcal{SN}}_\Omega$ iff $\text{Eval}_\Omega \in \widetilde{\mathcal{SN}}_\Omega$.*

This can be viewed as a completeness result about Eval_Ω, or a transfer theorem that tells when the transition from STEP A to STEP B in our 2-stage scheme has guaranteed success. In numerical computation, we have a "transfer process" that is widely used: suppose M is a real pointer machine that Ω-solves some semi-numerical problem $f : \mathcal{G}(\mathbb{R}) \dashrightarrow \mathcal{G}(\mathbb{R})$. Then we can define a numerical pointer machine $\widetilde{M}$ that computes $\widetilde{f} : \mathcal{G}(\mathbb{F}) \times \mathbb{F} \dashrightarrow \mathcal{G}(\mathbb{F})$, where $\widetilde{M}$ simply replaces each algebraic operation $o(w_1, \ldots, w_m)$ by its approximate counterpart $\widetilde{o}(w_1, \ldots, w_m, p)$ where p specifies the precision argument for $\widetilde{f}(G, p)$. In fact, it is often assumed in numerical analysis that STEP B consists of applying this transformation to the ideal algorithm M from STEP A. We can now formulate a basic question: under what conditions does $\lim_p \widetilde{f}(G, p) = f(G)$ as $p \to \infty$?

The framework in this section makes it clear that our investigation of real approximation is predicated upon two choices: base reals $\mathbb{F}$, and computational basis Ω. Therefore, the set of "inaccessible reals" is not a fixed concept, but relative to these choices. When a real pointer machine uses primitive operations $g, h \in \Omega$, we face the problem of approximating $g(h(x))$ in the numerical pointer machine that simulates it. Thus, it is no longer sufficient to only know how to approximate g at base reals, since $h(x)$ may no longer be a base real even if $x \in \mathbb{F}$. Indeed, function composition becomes our central focus. In the analytic and algebraic approaches, the composition of computable functions is computable. But closure under composition is not longer automatic for approximable functions. This fact might be initially unsettling, but we believe it confirms the centrality of the Eval_Ω problem, which is about closure of composition in Ω.

7 Conclusion: Essential Duality

Our main objective was to construct a suitable foundation for the EGC approach to real computation. Eventually, we modified the analytic approach, and incorporated the algebraic approach into a larger synthesis. In this conclusion, we remark on a recurring theme involving the duality between the algebraic and analytic world views, and between the abstract and the concrete sets.

The first idea in our approach is that we must use explicit computations. This follows Weihrauch's [39] insistence that machines can only manipulate names, which must be interpreted. Our intrinsic approach to explicit sets formally justifies the *direct* discussion of abstract mathematical objects, without the encumbrance of representations. This has the same beneficial effect as our underbar-convention (Section 2). Now, interpreting names is just the flip-side of the coin that says mathematical objects must be represented. In Tarski's theory of truth, we have an analogous situation of syntax and semantics. These live in complementary worlds which must not be conflated if they are to each play their roles successfully. Thus, semantics in "explicit real computation" comes from the world of analysis where we can freely define and prove properties of $\mathbb{R}$ *without asking for their effectivity.* Syntax comes from the world of representation elements and their manipulation under strong constraints. Interpretation, which connects these two worlds, comes from notations.

A similar duality is reflected in our algebraic-numeric framework of Section 6: STEP A occurs in the ideal world of algebraic computation, STEP B takes place in the constructive world of numerical computation. A natural connection between them is the transfer process from ideal programs to implementable programs. The BCSS manifesto [5] argues cogently for having the ideal world. We fully agree, only adding that we must not forget the constructive complement to this ideal world.

The second idea concerns how to build the constructive world for real computation: it is that we must not take the "obvious first step" of incorporating *all* real numbers. Any computational model that incorporates this uncountable set $\mathbb{R}$ must suffer major negative consequences: it may lead to non-realizability (as in the algebraic approach) or a weak theory (as in the analytic approach that handles only continuous functions). The restriction to continuous functions is unacceptable in our applications to computational geometry, where all the interesting geometric phenomena occurs at discontinuities. In any case, the corresponding complexity theory is necessarily distorted. We must not even try to embrace so large a set as the computable reals: the

Russian school did and paid the tremendous price of not being able to decide zero. Instead, we propose to only compute "approximations" in which all algorithmic input and output are restricted to well-behaved base reals. A natural and realistic complexity theory can now be developed. This complexity theory promises to be considerably more intricate than anything we have seen in discrete complexity. It is future work.

Consider the following natural reaction to our approximation approach: although we talk about real functions $f : \mathbb{R} \dashrightarrow \mathbb{R}$, our computational model only allows approximations, $\widetilde{f} : \mathbb{F} \times \mathbb{F} \dashrightarrow \mathbb{F}$. Why not simply identify "real functions" with the partially explicit functions of the form $\widetilde{f}$? This suggestion ("it would be more honest") is wrong for a simple reason: we *really* do wish to study the real functions f. All the properties we hold important are about f, not $\widetilde{f}$. Indeed, the analytic properties of $\widetilde{f}$ seems rather meager, and dependent on $\mathbb{F}$. If we discard f, and $\widetilde{f}$ is all we have, then whenever $\mathbb{F}$ changes we would be studying new functions, which is not our intention. Or again, consider our transfer theorem concerning the inclusion $\mathcal{SN} \subseteq \widetilde{\mathcal{SN}}$. Such an inclusion can only be considered because $\widetilde{\mathcal{SN}}$ is defined to comprise semi-numerical problems $f : \mathcal{G}(\mathbb{R}) \dashrightarrow \mathcal{G}(\mathbb{R})$ that are approximable. The "honesty" suggestion would be to equate $\widetilde{\mathcal{SN}}$ with the set of approximations $\widetilde{f} : \mathcal{G}(\mathbb{F}) \dashrightarrow \mathcal{G}(\mathbb{F})$.

We must avoid the intuitionistic (or formalists') impulse to discard the ideal world, and say that only the constructive world is meaningful. Nor must we believe that, with proper tweaking in the ideal world alone, we can recapture the properties of the world of computational machines and limited resources. No, we need both these complementary worlds of real computation, and fully embrace the essential gap between them. We can never exhaust the inaccessible reals, even though new advances in transcendental number theory continually make more reals accessible. This gap and tension is good and important: real mathematical progress is achieved at this interface.

Acknowledgments.

I thank Andrej Brauer, Klaus Weihrauch and Martin Ziegler for discussions in Dagstuhl about real computation. Also, thanks to Vikram Sharma, Sung-il Pae and Sungwoo Choi for feedback on the matter of this paper. One referee pointed out the connection of Mal'cev's work on numbering of sets and algebraic systems. I am deeply grateful for the thoughtful comments of Volker Bosserhoff and another anonymous referee; they both caught a serious error in my original characterization of computable real functions. They also provided half the proof of Lemma 23.

References

[1] A.Eigenwillig, L. Kettner, W.Krandick, K.Mehlhorn, S.Schmitt, and N.Wolpert. A Descartes algorithm for polynomials with bit stream coefficients. In *8th Int'l Workshop on Comp.Algebra in Sci.Computing (CASC 2005)*, pages 138–149. Springer, 2005. LNCS 3718.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[3] E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the complexity of numerical analysis. In *Proc. 21st IEEE Conf. on Computational Complexity*, 2006. To appear.

[4] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer, Berlin, 1985.

[5] L. Blum, F. Cucker, M. Shub, and S. Smale. Complexity and real computation: A manifesto. *Int. J. of Bifurcation and Chaos*, 6(1):3–26, 1996.

[6] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[7] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems*. American Elsevier Publishing Company, Inc., New York, 1975.

[8] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *J. of the ACM*, 23:242–251, 1976.

[9] R. P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Proc. Symp. on Analytic Computational Complexity*, pages 151–176. Academic Press, 1976.

[10] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Series of Comprehensive Studies in Mathematics, Vol.315. Springer, Berlin, 1997.

[11] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact efficient geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.

[12] E.-C. Chang, S. W. Choi, D. Kwon, H. Park, and C. Yap. Shortest paths for disc obstacles is computable. *Int'l. J. Comput. Geometry and Appl.*, 16(5-6):567–590, 2006. Special Issue of IJCGA on Geometric Constraints. (Eds. X.S. Gao and D. Michelucci).

[13] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[14] J. Demmel. The complexity of accurate floating point computation. Proc. of the ICM, Beijing 2002, vol. 3, 697–706.

[15] J. Demmel, I. Dumitriu, and O. Holtz. Toward accurate polynomial evaluation in rounded arithmetic, 2005. Paper ArXiv:math.NA/0508350, download from http://lanl.arxiv.org/.

[16] Z. Du, V. Sharma, and C. Yap. Amortized bounds for root isolation via Sturm sequences. In D. Wang and L. Zhi, editors, *Proc. Internat. Workshop on Symbolic-Numeric Computation*, pages 81–93, School of Science, Beihang University, Beijing, China, 2005. Int'l Workshop on Symbolic-Numeric Computation, Xi'an, China, Jul 19–21, 2005.

[17] A. Fabri, E. Fogel, B. Gärtner, M. Hoffmann, L. Kettner, S. Pion, M. Teillaud, R. Veltkamp, and M. Yvinec. The CGAL manual. 2003. Release 3.0.

[18] A. Fröhlich and J. Shepherdson. Effective procedures in field theory. *Philosophical Trans. Royal Soc. of London. Series A, Mathematical and Physical Sciences*, 248(950):407–432, 1956.

[19] P. R. Halmos. *Naive Set Theory*. Van Nostrand Reinhold Company, New York, 1960.

[20] J.V.Tucker and J.I.Zucker. Abstract computability and algebraic specification. *ACM Trans. on Computational Logic*, 3(2):279–333, 2002.

[21] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[22] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computation. In *Proc. 12th European Symp. on Algorithms (ESA'04)*, pages 702–713. Springer, 2004. Bergen, Norway. LNCS No.3221.

[23] K.-I. Ko. *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.

[24] C. Kreitz and K. Weihrauch. Theory of representations. *Theoretical Computer Science*, 38:35–53, 1985.

[25] B. Lambov. *Topics in the Theory and Practice of Computable Analysis*. Phd thesis, University of Aarhus, Denmark, 2005.

[26] A. I. Mal'cev. *The Metamethematics of Algebraic Systems. Collected papers: 1937–1967*. North-Holland, Amsterdam, 1971. Translated and edited by B.F. Wells, III.

[27] K. Mehlhorn and S. Schirra. Exact computation with `leda_real` – theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 163–172, Vienna, 2001. Springer-Verlag.

[28] N. Mueller, M. Escardo, and P. Zimmermann. Guest editor's introduction: Practical development of exact real number computation. *J. of Logic and Algebraic Programming*, 64(1), 2004. Special Issue.

[29] N. T. Müller. Subpolynomial complexity classes of real functions and real numbers. In L. Kott, editor, *Proc. 13th Int'l Colloq. on Automata, Languages and Programming*, number 226 in Lecture Notes in Computer Science, pages 284–293. Springer-Verlag, Berlin, 1986. I cite this paper for Weihrauch's broken arrow notation for partial functions... apparently, it is the older of the two notation from Weihrauch!

[30] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[31] M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer, Berlin, 1989.

[32] Research Triangle Park (RTI). Planning Report 02-3: The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), U.S. Department of Commerce, May 2002.

[33] D. Richardson. How to recognize zero. *J. of Symbolic Computation*, 24:627–645, 1997.

[34] D. Richardson and A. El-Sonbaty. Counterexamples to the uniformity conjecture. *Comput. Geometry: Theory and Appl.*, 33(1 & 2):58–64, Jan. 2006. Special Issue on Robust Geometric Algorithms and its Implementations, Eds. C. Yap and S. Pion. To appear.

[35] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[36] A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.

[37] D. Spreen. On some problems in computational topology. Schriften zur Theoretischen Informatik Bericht Nr.05-03, Fachberich Mathematik, Universitaet Siegen, Siegen, Germany, 2003. Submitted.

[38] B. L. van der Waerden. *Algebra*, volume 1. Frederick Ungar Publishing Co., New York, 1970.

[39] K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.

[40] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.

[41] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[42] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.