Constructive analysis, types and exact real numbers

Herman Geuvers, Milad Niqui, Bas Spitters, Freek Wiedijk
Radboud University Nijmegen, NL

December 16, 2005

Abstract

In the present paper, we will discuss various aspects of computable/constructive analysis, namely semantics, proofs and computations. We will present some of the problems and solutions of exact real arithmetic varying from concrete implementations, representation and algorithms to various models for real computation. We then put these models in a uniform framework using realisability, opening the door for the use of type theoretic and coalgebraic constructions both in computing and reasoning about these computations. We will indicate that it is often natural to use constructive logic to reason about these computations.

1 Introduction

Computing with real numbers is usually done via floating point approximations; it is well-known that the build-up of the rounding off that is inherent in these computations can lead to catastrophic errors [Krä97]. As a first attempt to prevent this problem one may use interval arithmetic [Kea96]. A different approach to computing with real numbers is *exact real arithmetic* which provides a precision-driven approach to computation with real numbers [YD95]. Exact real arithmetic is motivated by the need for unbounded precision in numerical calculations. Real numbers are infinite objects of which arbitrary good *finite approximations* can be given. A computable function over the reals is given by an algorithm that given the desired accuracy of the output, asks for a sufficiently good approximation of the input to be able to compute the result. Domain theory provides a systematic approach to interval computations and exact real arithmetic, using the higher order features of modern programming languages. A related, but slightly more concrete approach is Weihrauch's Type Two theory of Effectivity (TTE). In TTE one considers (Turing machine) computations on streams. Yet another approach (Markov's) is to use the function type $\mathbb{N} \to \mathbb{N}$, which is present in functional languages, to work directly with Cauchy sequences.

We will illustrate the spectrum between floating point computation and exact real arithmetic with a small example. Exact real arithmetic is mainly useful when one wants to answer precise mathematical questions by means of computation, and therefore we will use an example from mathematics.

Define a sequence of real numbers by iterating the *logistic map*:

$$x_0 = 0.5 \ , \ x_{n+1} = 3.999 \times x_n(1 - x_n) \ .$$

(Note that the number 3.999 should not be taken to be an approximation of some number from the real world, but should exactly have this value.) Now we want to determine a good approximation of the 10000th element in this sequence, x_{10000}. There are four ways to proceed with this, of increasing sophistication:

1. First we can use floating point arithmetic, using the IEEE 754 numbers implemented [IEE85] in the floating point unit of our computer. For instance, we might run the following small C program.

   ```
   main() {
   int n; double x = 0.5;
   for (n = 0; n < 10000; n++)
   x = 3.999 * x * (1 - x);
   printf("%f\n", x);
   }
   ```

   This will then give the output 0.780738. Now this number is totally unrelated to the correct answer, which (rounded to 6 decimals) is 0.354494. The sequence that the program calculates, because of rounding errors and the chaotic nature of the logistic map, will after the first few terms become essentially unrelated to the actual values of the sequence. This also becomes very clear if one runs the same program on Intel hardware which does not exactly follow the IEEE 754 standard. In that case, the program will print 0.999336.

   Note that interval arithmetic implemented using floating point numbers for the bounds does not help here. In that case the result of the program will be the interval $[0, 0.99975]$. While mathematically correct, this is not very informative.

2. The second approach to this problem, which *will* give the correct answer, is to use the methods of numerical analysis. One still calculates using floating point numbers, but with a greater precision. This is the method that the Maple computer algebra package uses. In the case of this example it turns out that calculating using 10^{4^2} digits will give the correct answer for x_{10^4}.

   Note that with this approach the numerical analyst is the one that will need to determine the necessary precision, this will not be automatically done by the computer. For this specific problem determining this precision might not be very difficult, but for more involved problems it might

easily become the time bottleneck in obtaining the answer (instead of the computation time by the computer.) Also, if this numerical analyst makes a mistake in his error estimates without this being noticed, then the answer will be incorrect without this being noticed. Therefore the correctness of the answer will not only depend on the correctness of the calculation software, but also on the correctness of the way that one poses the question.

3. The third approach for this problem is to have the computer keep track of the errors when running the calculation, and then have it rerun the program using a larger precision as long as the precision of the output is not good enough. That way the computer instead of the human will be the one to determine what precision will be needed.

   This approach can both be implemented using interval arithmetic (using bounds with sufficiently large precision), as well as using sufficiently precise floating point numbers together with an error estimate. It will be clear that both methods are essentially the same.

   However, with this approach we will have various intervals in our program that are related in the sense that they correspond to the same exact real number, but without this correspondence necessarily being reflected in the organisation of the program. This makes programming using this method more difficult than necessary.

4. The fourth approach is similar to the previous one, but this time one uses a functional ('higher order') data-type. Instead of using the ('first order') data-type of intervals one then uses functions that map desired precisions to intervals. That way the intervals that correspond to the same real number, but with different precisions, all become part of one data object in the program.

   Note that when using this approach it is important to *cache* the intervals that all these functions calculate. Otherwise the same interval might be recalculated many times, leading to a very bad complexity.

When looking at these four approaches, it will be clear that the last three all calculate the correct answer, and that they all use a similar amount of running time. This means that the increase of sophistication between the various methods that we have described does not correspond to a more efficient way of obtaining the answer to the problem. Instead it is primarily an improvement in *correctness*: the ease of getting a correct answer, and the ease of establishing that this answer is indeed correct.

Several approaches to exact real arithmetic have been implemented, as described in Section 2.2 below. An interesting question is what are the applications of these programs. As in the real world all data is inherently imprecise, it can be disputed whether for real-world applications exact real arithmetic will be an essential improvement over floating point computation. However, for mathematics its usefulness seems quite clear.

This has also raised the question of correctness of algorithms for real arithmetic. In some modern systematic approaches to program correctness one uses a realisability interpretation to get a precise and tight connection between proofs and programs. It turns out that the same can be done here. TTE may be seen as realisability using the second Kleene algebra. This means that there is a clear notion of an internal logic to reason about such computations. As usual when reasoning about computations, this internal logic is constructive. We will expand on this in section 3.5.

It should be noted that in the transition from floats to a language for exact real arithmetic with data types there is the usual friction between craft and technology: should these issues be treated carefully on an individual basis, or do we use the apparatus of domain theory? A similar tension exists for proofs: do we treat them individually, or do we use the technology of category theory, realisability and constructive mathematics?

The paper is organised as follows. We will focus on three important aspects of computing with real number: computations, semantics and proofs. Section 2 discusses representations and implementations. Section 3 discusses domain theory, Markov's recursive analysis, Type Two Effectivity, coalgebras and realisability. Section 4 contains type theory, program extraction and constructive analysis. We finish with a short conclusion.

2 Computations

2.1 Representations

While in theoretical models of computation real numbers can be considered as anything between a subset of the rationals and an object in a category, when it comes to practical computations, we must have a representation of real numbers (or of approximations to real numbers) that is easily understood by humans and computers. Usually this boils down to representing real numbers with decimals or bits; even though the intermediate steps can use other representations, the syntax for input and output real numbers (or approximations of real numbers) should not be far from the standard representations used in practice.

This brings up a serious problem: it is well-known that the standard decimal representation is not suitable for real computations. When multiplying the stream $x = 0.333\ldots$ (considered as an infinite input) by 3, the multiplication algorithm cannot give the first digit of the output: there is no way of deciding whether eventually the digit 2 may come (and then the first digit should be 0) or eventually a 4 may come (and then the first digit should be 1). Therefore, with the standard decimal representation, the deadlock is inevitable in calculating the outcome of multiplication, while multiplication is universally considered to be a computable function. This implies that the standard decimal representation is not computationally suitable. This shortcoming of the decimal representation was already known to Brouwer, who in [Bro21] showed — by means of a so-called weak counter example — that there are real numbers with no standard

decimal representation. In modern terms one might state this result as there is no computable map from, say, the Cauchy representation to the decimal representation of the real numbers. Or as we will express it in Section 3.3, the decimal representation is not admissible. As another consequence of the example above we see that the real numbers do not allow an effective way to compare real numbers, since the problem above arises precisely because we can not decide whether $x < \frac{1}{3}$ or $x \geq \frac{1}{3}$. Similarly, one does not have an effective equality test.

With the advent of the computers, and partly due to this theoretical shortcoming, partly in order to have a more efficient and hardware-compatible internal representation, other representations for real numbers were considered. Some of these non-standard representations were already known for centuries and others were discovered and further developed by computer scientists in the course of the 20th century. Knuth [Knu97, § 4] gives a thorough historical account of various representations for different number systems (integers, rationals and real numbers). While we do not intend to mention all the different representation systems we focus on some of the main ideas that are theoretically and practically important. Furthermore, while a study of different representations for integers and rational numbers is relevant for approximative computations with real numbers, since we are interested in exact real arithmetic we focus on the various approaches for representing real numbers.

Even though there are plenty of different representations for real numbers, in the broader context of computable analysis they can be seen as an instance of one of the few basic approaches. For example many representations that are used as basis of exact arithmetic implementations are based on the Cauchy sequences. It is fine-tuning of the details (such as modulus of convergence, or the representations for rational numbers) that makes the difference between such implementations.

Here we mention the three main classes of representations. Related (but different) classifications can be found in [WK87, Wei00, GL01]:

1. **Cauchy Sequences.** Cauchy sequences are traditionally the way that real numbers are represented in mathematics. In this approach real numbers are represented by Cauchy sequences of rational numbers (or some other dense countable Archimedean subset of the real numbers such as the dyadic numbers). The real number described by this sequence is the limit under the usual Euclidean metric. The most general case is the *naïve Cauchy representation* in which there is no modulus of convergence required. Although this is quite inefficient, its theoretical importance and its suitability for formalisation has made this representation to be the basis of the first full implementation of constructive real numbers in a proof assistant and have been used in formalising the proof of the fundamental theorem of algebra [GN02]. Other constructive formalisations which exist in the literature usually use a modulus of convergence [TvD88a, BB85, WK87].

   An important variation on the theme of Cauchy sequences are the so called

functional representations. In this approach real numbers are represented by functions on some fixed countable set, where the codomain of the function (the elements of the sequence) need not be rational numbers. The semantics of such a function is always (in some way) a Cauchy sequence, but the choice of a different domain or codomain can improve the representation. An example of such representation was proposed by Boehm et al. [BCRO86] where $\mathbb{Z}$ is used for both domain and codomain of the function representing a real number. Implicit semantic assumptions make sure that these sequences (which are not indexed by $\mathbb{N}$, rather by $\mathbb{Z}$) can be mapped to Cauchy sequences with a predetermined modulus of convergence. This approach forms the basis of several exact arithmetic packages, especially inside functional programming languages.

To give a specific example of this representation, if we represent the number $x_{10000} = 0.35449383309125298131\ldots$ which we defined on page 2, then in a decimal variant on Boehm's functional representation, three examples of possible representations for this number are

$$\begin{array}{rcr} & \vdots & \\ -1 & \mapsto & 0 \\ 0 & \mapsto & 0 \\ 1 & \mapsto & 3 \\ 2 & \mapsto & 35 \\ 3 & \mapsto & 354 \\ 4 & \mapsto & 3544 \\ & \vdots & \end{array} \qquad \begin{array}{rcr} & \vdots & \\ -1 & \mapsto & 0 \\ 0 & \mapsto & 0 \\ 1 & \mapsto & 4 \\ 2 & \mapsto & 35 \\ 3 & \mapsto & 354 \\ 4 & \mapsto & 3545 \\ & \vdots & \end{array} \qquad \begin{array}{rcr} & \vdots & \\ -1 & \mapsto & 1 \\ 0 & \mapsto & 1 \\ 1 & \mapsto & 4 \\ 2 & \mapsto & 36 \\ 3 & \mapsto & 355 \\ 4 & \mapsto & 3545 \\ & \vdots & \end{array}$$

and of course there are uncountably many more. These essentially are all maps that for each n gives n digits after the decimal point (either rounded up or rounded down).

2. **Dedekind cuts.** Dedekind cuts are an alternative approach to representing real numbers which is based on the least upper bound property of the reals rather than the Cauchy completeness. A key feature of Dedekind cuts, as compared to other representations, is their uniqueness: any real number is represented by precisely one cut. This feature, which is convenient to reason about cuts, makes it difficult to compute with them. In computational approaches to Dedekind cuts a set of rational numbers with additional computational structure is used to represent a real number which is the least upper bound (or greatest lower bound) of that subset; see [Wei00, p. 95] for details. Variations on this class of representations include choosing the characteristic function of a chosen dense subset of the reals such as dyadic numbers [WK87].

   Such representations have not been used for practical implementations but have been considered for reasoning about real numbers in mechanised reasoning. Again (as was the case with the naïve Cauchy representation)

this is due to the theoretical importance of these representations and their adaptability for use in formal mathematics. Examples of use of such representations include the formalisation of real numbers in the HOL theorem prover [Har94] and the formalisation of the real numbers that is used in the formalisation of the 4-colour theorem in the Coq proof assistant [Gon05].

3. **Streams of nested intervals.** The most standard way of representing real numbers is the decimal representation. This is a positional representation that falls under the more general case of radix representations in which a real number is represented by a stream of digits. In the base b radix representation (b-ary representation) starting from the first digit and moving to right, the effect of each digit can be seen as refining the interval containing the real number represented by the whole stream. Thus the b-ary representation can be seen as an instance of representing real numbers with a stream of shrinking nested intervals. Any such stream for which the diameter of the intervals converges to 0 represents a real number: the one that inhabits the infinite intersection of the intervals.

   If we return to our example, in a stream representation the number x_{10000} could be represented by the infinite stream of intervals:

$$\begin{gathered}\vdots\\ [-10, 10]\\ [-1, 1]\\ [0.3, 0.5]\\ [0.34, 0.36]\\ [0.353, 0.355]\\ [0.3544, 0.3546]\\ \vdots\end{gathered}$$

   These intervals correspond to the second functional representation on page 6. The other representations there also correspond to a (different) stream of intervals.

   By considering each interval as the image of the previous interval under a continuous real function one can encode the whole nested collection as an infinite composition of real maps applied to a base interval. This can be made formal using the following definition [Niq04, § 5.3].

Definition 2.1 (Generalised Digit Set) Let I be a closed subinterval of the compactification of the real numbers $[-\infty, +\infty]$. A set Φ of continuous functions on I is a *generalised digit set for interval* I if there exists a total and surjective map $\rho\colon \Phi^\omega \longrightarrow I$ (note that Φ^ω denotes the set of streams of Φ) such that for all streams $f_0 f_1 \cdots \in \Phi^\omega$ we have

$$\{\rho(f_0 f_1 \ldots)\} = \bigcap_{i=0}^{\infty} f_0 \circ \cdots \circ f_i(I) \ .$$

In other words, if each element x of I is the solitary element of some infinite composition of elements of Φ and each infinite composition of elements of Φ is a singleton. We call each element of Φ a *digit*.

The various representations of this family are characterised by the different restrictions that are put on the choice of the digits. In practice usually a finite set of Möbius maps satisfying some property is chosen as the set of digits while in literature the larger class of d-contractions is also studied [Kon00]. *Möbius maps* are maps of the form

$$x \longmapsto \frac{ax+b}{cx+d} \ ,$$

where $a, b, c, d \in \mathbb{C}$ and $ad-bc \neq 0$. In the context of stream representation for real numbers the Möbius maps with integer coefficients — also known as *linear fractional transformations* (LFT) or *homographic maps*— are considered. Taking the Möbius map with integer coefficients to be I-refining (i.e. mapping the closed interval I to itself) forms the basis of Edalat and Potts' approach to lazy exact arithmetic [EP97, Pot98, EPS99]. Restricting this further by taking $c = 0$ one arrives at representations by a finite set of *affine maps*, a subclass which includes the standard b-ary representations.

As stated the standard representation is not computationally suitable but the shortcoming is easily fixed by using one of the variants of radix representation. Different 'fixes' include changing the base to a non-integer base (eg. the golden ratio base [DG96]) or increasing the set of digits by introducing negative digits (eg. redundant b-ary representation [EP97]). Both these workarounds have been used long before the advent of computers. For example the introduction of negative digits is traced back to the number system implicit in the work of the 11th century mathematician Hasseb al-Tabari and even used in mechanical computing devices in the 19th century. A detailed historical survey can be found in [Knu97, pp. 205–208].

Our example number x_{10000}, when using a representation with negative digits, has many different representations. Next to its normal decimal representation

$$0.35449383309125298131\ldots$$

it can for instance also be written as

$$0.3545(-1)38331(-1)1253(-1)8131\ldots$$

It is also possible to use an infinite set of digits. The most important example where the set of digits is infinite is a representation based on continued fractions which we consider as a separate class of representations.

4. **Continued fractions** For centuries continued fractions have been used to represent rational and real numbers and elementary functions [Bre91]. As a result some implementations of exact real arithmetic are based on continued fraction representations. Some of such representation can be considered as a subclass of stream representations, but the most standard continued fraction representation (the so called $\mathbb{N}$-fraction) uses finite lists to represent rational numbers and streams for representing irrational numbers. The digits of the $\mathbb{N}$-fraction representation can be considered to be the Möbius maps of the form

$$x \longmapsto n + \frac{1}{x} \qquad n \in \mathbb{N} \ .$$

Taking finite lists and streams of such digits leads to a representation for real numbers larger than 1 that can be considered for exact arithmetic [Vui90, MM94]. One can allow for negative integers (and disallow 0,1 and -1) obtaining the $\mathbb{Z}$-fractions [Vui90, MM94]. In the $\mathbb{Z}$-fractions representation too, the rational numbers are represented by finite lists.

The above ($\mathbb{N}$-fraction and $\mathbb{Z}$-fraction) continued fraction representations serve well in the context of exact *rational* arithmetic; however for representing real numbers they suffer from the same computational shortcoming that exist for standard decimal representation. In order to overcome this some modifications of these representations have been studied and used for implementing exact arithmetic [Vui90, Les01]

There are several ways to obtain a representation based on continued fractions which uses infinite streams for rational and irrational numbers alike and therefore fall under stream representations. One way is to use the redundant Euclidean continued fractions [Vui90] which can be seen as an instance of Möbius maps [Pot98, § 7.3]. Another way would be to use the Stern–Brocot representation, again expressible in terms of Möbius maps [Niq04, § 5.7.1].

One can extend the notion of generalised digit set to include maps

$$\rho\colon \Phi^{\leq\omega} \longrightarrow I$$

where $\Phi^{\leq\omega}$ is the set of finite lists and (infinite) streams of elements of Φ to obtain a class of representations that includes the continued fraction representations. But since such representations are only considered in the context of continued fractions we preferred to classify them under a separate class.

The example number x_{10000} is a rational number, of which the $\mathbb{N}$-fraction

starts:

$$\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{7+\cfrac{1}{1+\cfrac{1}{1+\ \ldots}}}}}}}}}}}$$

The $\mathbb{Z}$-fraction of this number starts:

$$\cfrac{1}{3+\cfrac{1}{-6+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{8+\cfrac{1}{-2+\cfrac{1}{-7+\cfrac{1}{-3+\cfrac{1}{16+\ \ldots}}}}}}}}}}$$

In this case the coefficients come from the set $\{\ldots,-3,-2,2,3,\ldots\}$ instead of from the set $\{1,2,3,\ldots\}$. If one allows both negative numbers as well as the numbers 1 and -1 for the coefficients (this is needed to make the representation admissible), then this number gets many more representations.

2.2 Implementations

In recent years several approaches to exact real arithmetic have been implemented in various programming languages. Some of them have been considered for real-world applications especially in the field of visualisation and computational geometry [YD95, DEMY02]. There have also been studies to enhance the existing hardware architectures in the direction of exact real arithmetic [KM88, Men00].

There are three groups of implementations of exact real arithmetic. They are all based on the ideas that were presented above.

- The first group is an implementation of exact real arithmetic as part of a generic computer algebra package. Examples of such systems are commercial systems like Mathematica [Wol96] and Maple [MGH$^+$97]. These are

generally very fast at basic computations, but sometimes miss the features that are needed for involved problems.

- The second group are systems and libraries that have specifically been designed for exact real arithmetic, and that try to be as fast as possible at it. Examples of this are MPFR from LORIA [HLP+05], GiNaC/CLN by Kreckel [Kre05], iRRAM by Müller [Mül01, Mül05] and RealLib by Lambov [Lam05]. Programs from this group generally are implemented in C++.
- The third group are system that also have been designed for exact real arithmetic, but not for speed. Mostly these systems are part of an effort to move toward a provably correct implementation of exact real arithmetic. Examples of systems like this are Cr by Filliâtre [Fil05] (an ML reimplementation of CR, a Java system by Boehm [BCRO86, Sch89, Boe05]), ERA by Lester [Les05], Few Digits by O'Connor [O'C05], Bignum by Guy [Guy05] and ICReals by Edalat e.a. [PE97, Pot98, Eda05]. These systems generally are implemented in a functional programming language like ML or Haskell. Like the systems in the first group they generally miss the features that are needed to do advanced computations.

In practice the first group of systems is the fastest at basic problems, and the second group of systems are the only ones that are suitable for involved problems.

3 Semantics

3.1 Domain Theory

Domains are used to describe the semantics of programming languages, both the data types and the programs that are definable over them. They also provide a denotational model for computability, in the sense that the set of continuous functions from one domain to another is the mathematical counterpart to the set of computable functions (in some language or computational model). The basic structure in domain theory is that of a *directed-complete partial order* (dcpo), i.e. a partially ordered set in which every directed subset has a least upper-bound. A set A is *directed* if it is non-empty and every pair of its elements has an upper-bound in A; the least upper-bound of A is usually denoted by $\sqcup A$. Sometimes the notion of 'domain' is identified with that of a dcpo, but mostly, authors reserve 'domain' for a specific type of dcpo (with additional structure), depending on the application one has in mind, see [AJ94] for an overview on domain theory and the various notions of domains.

The interesting functions on dcpos are the *Scott continuous* ones: the monotone $f : D_1 \to D_2$ such that $f(\sqcup A) = \sqcup(f(A))$ for every directed set A (where the first $\sqcup$ is in D_1 and the second in D_2). The definable functions in a programming language (i.e. the computable functions in that language) are Scott continuous when interpreted as functions over a dcpo. The real 'power' of Scott

continuity lies in the fact that Scott continuous functions have a least fixed point. Thus, a recursive definition of a (functional) program can be given a meaning as the least fixed point of the (Scott continuous) functional it gives rise to.

The ordering on a dcpo is best viewed as an 'information ordering' or a 'definedness ordering'. A simple example is the *flat* dcpo of natural numbers $\mathbb{N}_\perp$, i.e. the set $\mathbb{N} \cup \{\perp\}$ made into a poset by letting $\perp \leq n$ for all $n \in \mathbb{N}$. Here, the elements are either 'totally defined' (we have full information on them) or they are 'totally undefined' (we have no information on them). A more interesting example is the set $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$ of Scott continuous functions from $\mathbb{N}_\perp$ to $\mathbb{N}_\perp$ ordered point-wise. The everywhere undefined function $\lambda x.\perp$ is the least element of this set and $f \leq g$ for functions f and g if g is 'at least as defined' as f in every element of $\mathbb{N}_\perp$. An important fact is that the set of Scott continuous functions between two dcpos forms a dcpo again.

The idea of 'approximation' is important in dcpos: *a approximates b* (or *a* is 'way below' *b*), notation $a \ll b$, if, whenever $b \leq \sqcup X$, then $a \leq x$ for some $x \in X$ (where X of course ranges over the directed subsets). An element a is *compact* (or *finite*) if $a \ll a$. The compact elements of a dcpo form an important set, which is usually written as $\mathsf{K}(D)$. For $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$, $f \ll g$ implies that f is defined only on a finite set of elements ($f(x) \neq \perp$ for finitely many x). The collection of the functions with a property like f are also the compact elements of this dcpo. A dcpo is *continuous* if there is a *basis* B: a set of elements such that $x = \sqcup\{y \in B \mid y \ll x\}$ for all x. Thus, in a continuous dcpo, all elements can be written as the lub of the basis elements that approximate it. A dcpo D is called *algebraic* if the set of its compact elements forms a basis. The adjective 'ω' is added to say that the basis is countable, as in ω-continuous dcpo and ω-algebraic dcpo. In [GS90], the notion of domain is identified with ω-algebraic dcpo. That ω-algebraic dcpos are of special interest comes from the fact that a continuous function $f : D \to E$ between two ω-algebraic dcpos can be fully characterised (in a countable way) by its compact elements as follows.

$$f(x) = \bigsqcup\{y \in \mathsf{K}(E) \mid y \leq f(x') \text{ for some } x' \in \mathsf{K}(D) \text{ with } x' \leq x\}$$

In the case of computability over the real numbers, some dcpos are of specific interest. One is the interval dcpo of nested intervals $I(\mathbb{R})$, consisting of $\mathbb{R}$ and the non-empty closed real intervals, ordered by reverse inclusion: $I \leq J$ if $I \supseteq J$. This domain was first proposed, in a slightly different form, by [Sco72]. The intervals should be understood as approximations of real numbers, so a smaller interval gives more information, $\mathbb{R}$ is the $\perp$-element of the information order and singleton intervals $\{a\}$ are maximal elements. For the 'way below' relation we have that $a \ll b$ iff b is a subset of the interior of a. The rational intervals together with $\mathbb{R}$ form a countable basis for $I(\mathbb{R})$. A directed subset of $I(\mathbb{R})$ is a collection of intervals A such that $\cap A \neq \emptyset$; the lub of such a directed set A is its intersection: $\sqcup A := \cap A$, which is a closed non-empty interval again. The nice thing about this dcpo is that it generalizes functions

from $\mathbb{R}$ to $\mathbb{R}$ in a simple way to incorporate partial functions, in such a way that partiality is not just undefinedness, but may involve some partial information (an interval approximation). A continuous function $f : \mathbb{R} \to \mathbb{R}$ extends in the straightforward way to a function $\hat{f} : I(\mathbb{R}) \to I(\mathbb{R})$; the other way around, a function $g : I(\mathbb{R}) \to I(\mathbb{R})$ represents a partial function $\bar{g}$ from $\mathbb{R}$ to $\mathbb{R}$ given by $\bar{g}(x) := y$ if $g(\{x\}) = \{y\}$ and undefined otherwise. See [EH02, EL04] for a more detailed study.

RealPCF, a programming language with real as a basic data type of [Esc96], was designed in such a way that the nested interval semantics would be its denotational semantics. An interesting feature of RealPCF is that it contains a parallel 'if' construct. It is shown in [EHS04] that this non-sequentiality is an inherent feature of the nested interval domain. See [ES99, EE96] for a further discussion of the expressivity of RealPCF.

A second example of a dcpo of interest in this context is the dcpo of finite and infinite lists over some given pointed dcpo. Typically, the pointed dcpo is $A_\perp$, the flat dcpo over some finite set A, where the elements of A are the 'digits'. The dcpo of finite and infinite lists is then denoted by $[A]$, and the intuitive understanding of an element $[a_1, a_2, \ldots, a_n]$ is that of a finite approximation of a real number. The representations based on streams of nested intervals are instances of this. The ordering on $[A]$ is the ordering on $A_\perp$ extended to a prefix ordering. Stated differently, $[A]$ is the solution to the domain equation $X = A_\perp \times X$ in the category of dcpos [Niq04, § 4.5].

The TTE model of computable real valued functions (see Section 3.3) uses as representation of the reals exactly the data-type $[A]$ (under the proviso that it's 'admissible' – see Definition 3.1 – which excludes the decimal representation). So in TTE a real number is a digit stream with digits from A and the TTE-computable functions are all continuous.

Similar to the interval domain, which is used as the semantics for programming in RealPCF, the domain $[A]$ can also be used as the semantics of programming with real numbers. This has been shown by [Sim98], who uses PCF (of [Plo77]) as a programming language and the data type of streams over $\{-1, 0, 1\}$ as the type of the reals. In [BES02], this approach is compared with the one of RealPCF and it is shown that the expressivity is the same up to second order types. An interesting aspect is that the PCF based approach is purely sequential. This is due to the fact that RealPCF uses real as an abstract primitive data type, so special primitive programming constructs are required to compare two reals, whereas in the PCF based approach one programs directly with the representations.

3.2 Markov's Recursive Analysis

Here we will shortly introduce Markov's constructive recursive mathematics (CRM). Historically, this seems to be the first concrete model for exact real computations. We refer to [TvD88b, BR87, Bee85, Abe80] for more information. In CRM one represents an object by a certain partial *recursive* function. In this way most of the representations above may be treated. For concreteness let us

consider the Cauchy representation of the real numbers. A real number x is thus represented by a partial recursive function which on input n returns the nth element of a Cauchy sequence with limit x. One thus computes not with all the real numbers, but only with the recursive ones. Fortunately, the well-known constants e, π, ϕ are recursive and so are the trigonometric functions. In this way one can develop exact real arithmetic in CRM.

Since there are only countably many codes for recursive functions there are only countably many (recursive) real numbers in CRM. This may be counterintuitive at first, however, given any recursive sequence of real numbers, Cantor's diagonal construction supplies a new recursive real number which is apart from all of them. Thus the recursive real numbers are countable, but recursively uncountable! In practice this last fact is more important and can be restated as 'the real numbers are uncountable' in the internal logic of CRM. We will introduce the internal logic in section 3.5.

In CRM all (recursive) functions between real numbers are continuous [KLS57a, KLS57b, Ceĭ62, Ceĭ59].

Theorem 3.1 (Kreisel, Lacombe, Schoenfield, Tsejtin) *In CRM: Every (total) mapping of a complete separable metric space into a metric space is continuous.*

The theorem is stated in the internal logic, meaning that all the objects should be presented effectively. We will discuss the internal logic more thoroughly in Section 3.5. This theorem may seem counterintuitive at first, when applied to the real function which is 0 for $x < 0$ and 1 for $x \geq 0$? This is not a total function in CRM — that is, we cannot recursively decide whether $x < 0$ or $x \geq 0$ for each real number x. Having such a test would solve the halting problem.

For concrete functions on the real numbers this model behaves as expected. However, when quantifying over compact spaces there are some surprises. For instance, one can define an unbounded continuous function on the unit interval. Such a function cannot be uniformly continuous. It is defined in the internal logic, however, externally one sees that such a function is defined on all the recursive points, but not on *all* points.

To avoid such problems Brouwer introduced his choice sequences [Bro75, Hey56, Tro77]. Kleene and Vesley captured much of his theory using a realisability interpretation. This interpretation was rediscovered by Weihrauch in his TTE, a theory that we will now discuss.

3.3 TTE

Among the many schools of computable analysis, most of which are known to be equivalent, TTE (Type Two Effectivity) is a theory of computability based on Turing machines with infinite input and infinite output [Wei97, Wei00]. The TTE notion of computability is nothing but computability of algorithms on infinite sequences. This is because both the input and output tape can be thought of as a stream (lazy infinite sequence), and hence the TTE model will be very

similar to the actual computation on the higher order data structure of streams or functional representations for real numbers (see Section 2.1). Therefore it is logical to consider computability for algorithms in exact arithmetic with respect to TTE. This is in contrast to other approaches to computability which usually involve heavy encoding of streams and finite lists. The intuitive model that TTE uses not only makes the programmer's understanding of the complexity of the algorithms relatively easy but also provides a notion of computability that directly works on the representations of real numbers. In fact the notion of representation plays a central rôle in TTE, providing a good framework to compare the relative theoretical strength of various representations.

In this sense one can use TTE to give a formal explanation of the shortcomings of the standard decimal representation which was mentioned in the example in Section 2.1. There we showed that — informally— the standard decimal representation is not 'computationally suitable'. In TTE there is a notion of admissibility of representations, which rigorously defines whether a representation of real numbers is computationally suitable.

Definition 3.1 (Admissible Representation) Let I be a closed subinterval of the compactification of real numbers $[-\infty, +\infty]$. Let Φ be a set (finite or infinite) of digits and Φ^ω be the set of streams of elements of Φ. A map $p\colon \Phi^\omega \longrightarrow I$ is an *admissible representation* of I if the following conditions hold.

1. p is continuous with respect to the product topology of the discrete topology on Φ;
2. p is surjective;
3. p is *maximal*, i.e., for every (partial) continuous $r\colon \Phi^\omega \longrightarrow I$, there is a continuous $f\colon \Phi^\omega \longrightarrow \Phi^\omega$ such that $r = p \circ f$.

The notion of admissibility relates the notion of computability on streams with continuity on real numbers. Intuitively an admissible representation gives rise to functions which are computable with type two Turing machines. Obviously, the standard decimal representation turns out to be not admissible. In fact following the example in Section 2.1 one can show that the multiplication by 3 is not a TTE-computable function when using the standard decimal representation [Wei97].

An important property of admissible representations is that they provide a redundant representation for real numbers. This means that every real number has several representations. Examples of admissible representations include the redundant b-ary representation for $[0, +\infty]$ used in [EP97], the ternary Stern–Brocot representation for $[0, +\infty]$ [HN05] and the binary Golden ratio notation for $[0, 1]$ [DG96].

As stated before, the TTE notion of computability that is based on the admissible representations is equivalent to most other models of computability [Wei00, § 9].

3.4 Coalgebras

Coalgebras (also called 'systems' in [Rut00, BM96]) provide a semantics for structures that can be considered as an infinite process, of which only partial observations are available. Examples of such structures are real numbers, labelled transition systems, object oriented modularity and dynamical systems. A modern survey of coalgebras and their applications can be found in [Jac05].

In the category theoretic semantics for computer science, to any functor corresponds a category of coalgebras of that functor. For certain functors, this category happens to have a final object. Those functors, or to be more precise, the final coalgebra of those functors, are used to model infinite processes.

Let $\mathcal{C}$ be a category, and F be an endofunctor on $\mathcal{C}$. A F-*coalgebra* is a pair $\langle Y, y\rangle$ in which Y is an object of $\mathcal{C}$ and $y\colon Y \longrightarrow \mathrm{F}(Y)$ is a morphism in $\mathcal{C}$. We call the first element of the pair the *carrier* of the coalgebra, and the second element of the pair the *structure map* of the coalgebra.

Let $\langle U, u\rangle$ and $\langle V, v\rangle$ be F-coalgebras. Then a *coalgebra map* from $\langle U, u\rangle$ to $\langle V, v\rangle$ is a map $f\colon U \longrightarrow V$ such that $v \circ f = \mathrm{F}(f) \circ u$; i.e., the following diagram commutes.

$$\begin{array}{ccc} U & \xrightarrow{\quad f \quad} & V \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle v} \\ \mathrm{F}(U) & \xrightarrow[\mathrm{F}(f)]{} & \mathrm{F}(V) \end{array}$$

It can be checked that the identity morphism is a coalgebra map, and that the composite of two coalgebra maps is again a coalgebra maps. Hence the F-coalgebras form a category. We are particularly interested whether this category has a final object. If such a final object exists, we assign some fixed notation to it.

Definition 3.2 (Final Coalgebra, Coiterator) A *final* F-*coalgebra* is a coalgebra $\langle \nu\mathrm{F}, \nu\text{-}\mathsf{out}\rangle$ such that for every coalgebra $\langle U, u\rangle$ there exists a unique coalgebra map $\nu\text{-}\mathsf{it}(u)$ from $\langle U, u\rangle$ to $\langle \nu\mathrm{F}, \nu\text{-}\mathsf{out}\rangle$. We shall call $\nu\text{-}\mathsf{it}(u)$ the *coiterator* of u.

Thus a coalgebra $\langle \nu\mathrm{F}, \nu\text{-}\mathsf{out}\rangle$ is final if and only if

$$\forall U\colon \mathcal{C},\ \forall u\colon U \to \mathrm{F}(U),\ \exists!\, \nu\text{-}\mathsf{it}(u)\colon U \to \nu\mathrm{F},\ \ \nu\text{-}\mathsf{out} \circ \nu\text{-}\mathsf{it}(u) = \mathrm{F}(\nu\text{-}\mathsf{it}(u)) \circ u\ ;$$

equivalently if the following diagram commutes.

$$\begin{array}{ccc} U & \overset{!\nu\text{-}\mathsf{it}(u)}{\cdots\cdots\cdots\cdots\cdots\cdots\succ} & \nu\mathrm{F} \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle \nu\text{-}\mathsf{out}} \\ \mathrm{F}(U) & \xrightarrow[\mathrm{F}(\nu\text{-}\mathsf{it}(u))]{} & \mathrm{F}(\nu\mathrm{F}) \end{array}$$

In an arbitrary category, the question whether a final coalgebra for a given functor exists is not always easy to answer. However, if a final F-coalgebra exists then it is unique up to isomorphism. Moreover the structure map of a final coalgebra is an isomorphism; hence, a final coalgebra is a fixed point for its functor [JR97]. This fixed point property is the origin of our interest in a final coalgebra, because it makes it possible to use the theory of final coalgebras as a semantics for data types of infinite objects.

Finality of coalgebras provides us with coinductive proof principles, which can be used to reason about objects residing in a final coalgebra. A well-known example of a coinductive proof principle is the notion of bisimulation. This can roughly be stated as: two infinite processes are equal if they are bisimilar, i.e. if the observable parts are equal and the continuation of the two processes (or the subprocesses) are again bisimilar.

As an example we consider the set of *streams*. In lazy exact arithmetic, real numbers are represented by means of lazy infinite sequences of elements of a set, so called streams. The collection of streams of the elements of the set Φ is the final coalgebra of a simple polynomial functor, namely $\mathrm{F}(X) = \Phi \times X$ in the category **Set**. Taking as structure map the map $\langle \mathsf{hd}, \mathsf{tl} \rangle \colon \Phi^\omega \longrightarrow \Phi \times \Phi^\omega$ one can show that the set of streams is indeed a final coalgebra for F [Rut00]. The constructor of the final coalgebras of streams is $\mathsf{cons} \colon \Phi^\omega \longrightarrow \Phi^\omega$ which prepends an element to the beginning of a stream.

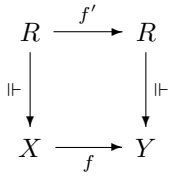
The standard decimal representation for real numbers is a stream representation: each real number is denoted by a stream over the 10-element set of digits. So are the various admissible representations that are used in exact real arithmetic. In this way one views the real numbers as a final coalgebra. In this setting the functions on real numbers become maps in the category of coalgebras, the so called coalgebra maps. This does not capture all functions on real numbers, but only those for which we have suitable partial observations, i.e. computable finite approximations. Hence, in order to present a theory of computable coalgebra maps one has to adhere to domain-theoretic (or equivalent TTE) approaches for a suitable definition of computability [Pat03]. A more structural solution would be to interpret coinductive types in a realisability model, as we will present in the next section.

3.5 Realisability

In this subsection we will describe realisability. For general overviews we refer to [vO02, Tro98]. After a general introduction to realisability we will shortly describe three realisability interpretations: for recursive analysis, for TTE and for domain theory. In this way we obtain a nice uniform treatment of the three models previously discussed. Our presentation of realisability models in this section mostly follows the presentations by Birkedal [Bir00] and Bauer [Bau00, Bau05] and we refer to these works and [vO02] for historical background.

In order to represent data on a computer we need to find a code, a realization, for it. This suggests a *realisability* relation, that is a relation $\Vdash$ between a set of codes R and a set X such that each code represents at most one element.

Functions are realized via the following commutative diagram.



It is then said that f' tracks f. These representation are connected to the representations discussed before. There is a one-one correspondence between a realisability relation $\Vdash$ and a partial function δ defined by

$$\delta a = x \qquad \text{iff} \qquad a \Vdash x.$$

Yet another equivalent presentation is given by partial equivalence relations. The relation 'to be a code for the same element' is a partial equivalence relation.

In order to be able to represent functions by our codes we should be able to interpret some applicative structure. It turns out that one requires the realizers to have the structure of a partial combinatory algebra (PCA). A PCA is a structure $(X, \bullet, \mathbf{k}, \mathbf{s})$ which has all the relevant properties of the combinator presentation of recursion theory. In Kleene's original realisability interpretation the realizers are the natural number encoding of the partial recursive functions. This prime example of a PCA is called the first Kleene algebra and is simply denoted $\mathbb{N}$. In fact, in this way we obtain the computational model of Markov's recursive mathematics. However, how do we include a data-type for all, not only the recursive real numbers? To phrase it differently, how does one make a realisability model corresponding to, say, TTE? To do this one needs a slightly different picture. The structures are realized by *all* streams — that is, elements of *Baire space*[1] — but we only allow *recursive* functions. To solve this one uses the notion of *relative realisability*, where one takes the data from A, but restricts the functions to a sub-PCA $A_\sharp$. In fact, this $\sharp$ may be seen as a modal operator on types, assigning to each type its subtype of 'computable' elements, see [Bir00].

The data types are captured by the notion of a *modest set* over a PCA A — that is, a set with a realisability relation $\Vdash$. The category of modest sets over A with functions over $A_\sharp$ is denoted by $\mathrm{Mod}(A, A_\sharp)$. When both the sets and the functions are represented by the same PCA A one simply writes $\mathrm{Mod}(A)$. Now given a definition of computability on real numbers, one may ask how to define computability of lists of real numbers, trees of reals numbers, streams of real numbers, the positive real numbers... Although one can give concrete answers for each particular model it would be better to have a structural solution that works for all these models at once. In advanced programming languages these issues are solved by the presence of a strong type system which is closed under certain type forming constructions, see Section 4.1. Categorical logic and type theory [LS88, Jac99] allow us to define a very strict and structural

[1] We have denoted Baire space before as Φ^ω when we choose the alphabet Φ to be the natural numbers.

connection between logic and semantics by the use of the *internal* logic of a (categorical) model. It is customary to speak about the internal logic when one is really talking about the internal logic *and* type theory. We will stick to this custom. It should be noted that the principles valid in the internal logic in general depend on the principles assumed to hold in the meta-logic. The internal logic for realisability using modest sets is intuitionistic logic, the logic of constructive mathematics. The internal type theory supports dependent, inductive and coinductive types. By providing realisability interpretations for all the models discussed before we show that we can use this type theory in all these models.

To give a flavour of how one realizes logic, we present the abstract realisability interpretation. Here x and y are elements of the PCA. The symbols '$x\underline{\mathbf{r}}P$' may be pronounced as x realizes P.

$$
\begin{aligned}
x\underline{\mathbf{r}}P &:= P \wedge x\downarrow \quad \text{for } P \text{ atomic}; && (1)\\
x\underline{\mathbf{r}}(A \wedge B) &:= (\mathbf{p_0}x\underline{\mathbf{r}}A) \wedge (\mathbf{p_1}x\underline{\mathbf{r}}B); && (2)\\
x\underline{\mathbf{r}}(A \to B) &:= \forall y(y\underline{\mathbf{r}}A \to x \bullet y\underline{\mathbf{r}}B) \wedge x\downarrow; && (3)\\
x\underline{\mathbf{r}}\forall yA &:= \forall y(x \bullet y\underline{\mathbf{r}}A); && (4)\\
x\underline{\mathbf{r}}\exists yA &:= \mathbf{p_1}x\underline{\mathbf{r}}A[y/\mathbf{p_0}x]. && (5)
\end{aligned}
$$

Here $\mathbf{p_i}$ denotes the ith projection of a pair. The symbol $\downarrow$ may be read as 'is defined'. We say that A is true in the model when A is realized, that is the set of realizers is inhabited. A similar definition for the realization of types can be given analogous to the Curry–Howard isomorphism which we will discuss in section 4.1.

We will continue to give the realisability interpretation for the three models presented before.

Markov's recursive mathematics

Markov's recursive mathematics can be modelled by Mod($\mathbb{N}$) the modest sets over the first Kleene algebra $\mathbb{N}$, that is the realizers in the PCA $\mathbb{N}$ are the ordinary (partial) recursive functions coded by natural numbers.

The internal logic of CRM satisfies not only the usual axioms of intuitionistic logic, but also Church's thesis and Markov's principle. Church's thesis states that we only work on recursive sequences. That is, our programming language allows us to access the codes of the real numbers.

$$\forall n\exists mA(n,m) \to \exists k\forall n\exists m[A(n,Um) \wedge Tknm] \ .$$

Here T denotes Kleene's T-predicate and U is the function that returns the result Um of the computation m. This variant of Church's thesis may be read as: if for each n we can find an m such that $A(n,m)$, then we can find a recursive function that finds such m for us. Markov's principle allows us unbounded search: if we know that an element with a decidable property can not fail to exist, we can just start searching until we find it. Let P be a decidable predicate.

$$\neg\forall n.\neg P(n) \to \exists n.P(n) \ .$$

As stated before this model behaves somewhat unexpectedly when quantifying over compact spaces. For instance, a point-wise continuous function on a compact interval may be unbounded. This is due to the failure of the fan-theorem, the constructive variant of König's lemma. In order to remedy this, one introduces choice sequences, a concept which is captured by Kleene–Vesley's realisability model which we will discuss now.

TTE

TTE is the model $\mathrm{Mod}(\mathbb{B}, \mathbb{B}_\sharp)$, the second Kleene algebra which was extensively studied by Kleene and Vesley. Thus TTE may be seen as the Kleene–Vesley realisability interpretation [KV65]. Troelstra [Tro92] seems to have been the first to observe the possibility to use realisability to obtain results in TTE.

It may seem surprising that the notion of admissible representation which is so important in TTE seems absent in realisability. To understand this let us consider a representation of the real numbers. First of all, there is no absolute pre-given notion of real number, thus it seems impossible to state what an admissible representation of 'the' real numbers is. However, one can axiomatically define the real numbers up to isomorphism. Now we fix any representation of the real numbers. We choose the Cauchy representation for definiteness. One defines a representation r of the real numbers as a surjective map from Baire space to the Cauchy real numbers. Of course, surjective should be interpreted in the internal logic. The Cauchy real numbers may be represented by a map $c : \mathbb{B} \to \mathbb{R}$. Thus surjectivity means

$$\forall \beta \in \mathbb{B} \exists \alpha \in \mathbb{B}[r(\alpha) = c(\beta)] \ .$$

By applying the axiom of choice for variables over Baire space— denoted C-C in [TvD88b] — we obtain:

$$\exists f : \mathbb{B} \to \mathbb{B} \forall \beta \in \mathbb{B}[r(f(\beta)) = c(\beta)] \ .$$

This is precisely the maximality condition in the notion of admissible representation defined above. Thus one may view admissibility as the *external* way of stating the surjectivity of a representation. We should mention that the axiom C-C which we used above holds in the internal logic.[2]

It has been crucial in the development of constructive mathematics that complete separable metric spaces can be represented by a continuous surjective image of Baire space. This same fact is also heavily used in the context of TTE. This allows us to directly transfer constructive theorems about such spaces to TTE, see [Lie04] for details. Similarly, compact metric spaces can be represented by Cantor space and a similar transfer principle exists.

To sum up, one can now view TTE as the assembly language for exact real computation. Using categorical logic and realisability, one can compile a dependently typed functional language with (co)inductive types into this Turing machine model. Thus the relation between constructive mathematics and

[2]We have been unable to find this simple remark in the literature.

TTE is much like the relation between an advanced programming language, say Haskell or ML, and an assembly language. The former provides more structure, the latter gives finer control over the computation and in particular over the complexity of such computations.

Domains

The theory of effectively presented continuous domains as used by Edalat and co-workers [Eda97] fits into the model $\mathrm{Mod}(\mathbb{P}, \mathbb{P}_\sharp)$. Here $\mathbb{P}$ denotes Scott's graph model [Sco76], which may be seen as the 'universal' countably based T_0 topological space; see [Bau00] for details.

Coinductive types

As mentioned before the realisability models support coinductive types. One way of seeing this is to observe that such models can be extended to a topos — a generalised, or local, set theory. This construction is due to Hyland [Hyl82] and is called the effective topos. Thus when we define our data-types coinductively we can directly interpret them in all the realisability models we have described.

As an example consider the coinductive streams of natural numbers. It is straightforward to prove constructively that this final coalgebra is the function space $\mathbb{N} \to \mathbb{N}$. Thus one can directly interpret these streams in all the models above. For instance, in CRM all these functions would be recursive.

4 Proofs

4.1 Type theory

Type theory provides a syntactic analysis of the notion of computability. In this section we describe some basic concepts of type theory that are relevant for understanding the connections with constructive analysis and computation with the reals. For more details on type theory we refer to [ML84, NPS90, Luo94, Bar92, BG01] (We will not deal with programming language aspects of type theory, for which we refer to [Pie02], nor shall we discuss logical frameworks, for which we refer to [Pfe01].) The basic notion of type theory is obviously that of a type, which describes a collection of terms (the terms of that type) in a *syntactic* way: there are rules for constructing terms of a type of a specific form (so called *introduction rules*) and there are rules for using terms of a type of a specific form (so called *elimination rules*). The crucial point is that it is *decidable* whether a term is of a given type, because the type of a term can be computed on the basis of its syntactic shape. (There are some exceptions, but almost all type theories adhere to this principle.) This distinguishes type theory from set theory: $X := \{n \in \mathbb{N} \mid \forall x, y, z,\ x^n + y^n \neq z^n\}$ is a typical example of a set and not a type (whether $n \in X$ is not a matter of syntactic analysis of n).

Simple examples of types are bool and nat. The type bool contains just true and false and nat contains 0 and, if $x : \mathsf{nat}$, then $S\,x : \mathsf{nat}$ as well. Apart from

construction principles for terms, there are construction principles for types as well, for example, given the types σ and τ, we have $\sigma \times \tau$ and $\sigma \to \tau$ as types, with the associated construction principles of 'pairing' and λ-abstraction. This gives rise to the system $\lambda{\to}{\times}$ of simple type theory with products. Exactly how much information one puts in the terms (and in what form) is a matter of choice. For programming purposes, one usually would want to put as little information as possible (because the program is what the user writes) and let the computer (compiler) compute a type (or a set of types) for us. So, for $\lambda{\to}{\times}$, one can have as construction rule that $\langle M, N\rangle : \sigma \times \tau$ if $M : \sigma$ and $N : \tau$ and that $\lambda x.M : \sigma \to \tau$ if $M : \tau$ under the assumption that $x{:}\sigma$.

The construction and elimination principles of type theory give it a strongly constructive flavour, which was first made explicit by Martin-Löf: we describe a collection by telling how we can construct objects of that collection. Due to the fact that we know the construction principles, we can define a function *from* the collection by distinguishing cases according to the construction rules (and doing a recursive call if needed).

There are many different type theories, depending on the types one allows and the functions one allows to define over them. Examples of additional type constructions are: polymorphic types, higher order polymorphic types, dependent types, inductive types and recursive types. An important aspect of the definable functions in type theory is that they are executable, due to the computational model of the λ-calculus that is part of the system.

4.1.1 Curry–Howard Isomorphism

Apart from a computational model, type theory also incorporates a 'logical model'. This is due to the Curry–Howard–de Bruijn isomorphism, that interprets formulas as types and proofs (logical deductions) as terms. The isomorphism was first noticed by Curry for minimal propositional logic and simple type theory, and later extended to the first order case by [How80] (but the original paper dates back to 1968). Howard who also treats the case of proofs by induction over the natural numbers and he has also coined the name 'formulas-as-types'. Independently of Howard, De Bruijn noticed the formulas as types analogy in the late 60's in the context of his logical framework Automath [Bru80]. In the analogy of De Bruijn, the logic is encoded in type theory, so his formulas-as-types analogy is slightly different from what we discuss here. (As a matter of fact, various encodings of logic in type theory were studies, and some the later ones are quite close to what we treat here.) The isomorphism can also be seen as an operationalisation of the so called BHK (Brouwer–Heyting–Kolmogorov) interpretation of proofs, where e.g. a proof of $A \to B$ is interpreted as a *method* for producing a proof of B out of a proof of A; see Section 4.3 for details. This was also the interest of Martin-Löf in the formulas-as-types isomorphism, who took it as the starting point of his intuitionistic theory of types [ML84] and extended the isomorphism to the existential quantifier and inductive types.

Combining the computational and the logical interpretation of type theory,

we find that the basic judgement

$$\Gamma \vdash M : A$$

can have two 'readings':

1. M is a piece of data (or algorithm) of data type A,
2. M is a proof (deduction) of formula A.

To make a (syntactic) distinction between data types and formulas, most type theories have (at least) two 'universa' or 'sorts': Set and Prop, where $A : \mathsf{Set}$ means that A is a data type and $A : \mathsf{Prop}$ means that A is a formula. The context Γ consists of variable declarations $x : B$ and definition $c := t : B$. Variable declarations are read as assumptions (assuming a hypothetical proof of B) in case $B : \mathsf{Prop}$. A definition is read as a reference to a proved lemma (with proof t) in case $B : \mathsf{Prop}$.

The correspondence between logic and type theory is so strong that there is an isomorphism between logic (e.g. the $\wedge\!\rightarrow$-fragment of propositional logic) and type theory (the system $\lambda\!\rightarrow\!\times$). The isomorphism maps formulas to types and proofs in natural deductions to terms. In this isomorphism, the logical introduction rules correspond to the construction principles of the type theory and the logical elimination rules correspond to the elimination principles. The isomorphism also maps computations in logic (via cut-elimination) to computations in type theory (e.g. β-reduction in $\lambda\!\rightarrow\!\times$).

To extend the Curry–Howard isomorphism to predicate logic, we need 'formula types' (types of type Prop) that depend on objects of a 'data type' (a type of type Set). A predicate over the type nat should be a function from nat to Prop and similarly, a binary relation (like $\leq$) should be of type $\mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{Prop}$. This phenomenon is called *type dependency*: the possibility to form type expressions that contain term expressions as subterms. Type dependency also implies the formation of the *dependent typed function space*, usually written as $\Pi x{:}A.B(x)$, denoting the type of functions that takes an $a : A$ and produces a term of type $B(a)$. These dependent function types are typically used for formalising the $\forall$ quantifier: a proof of $\forall x{:}A.B(x)$ is a method that, given an $a : A$ produces a proof of $B(a)$. Similarly one can also introduce a type dependent product type $\Sigma x{:}A.B(x)$. This type consists of pairs $\langle a, b\rangle$ where $a : A$ and $b : B(a)$. There are various choices for the elimination rule for Σ-types, the simplest being: if $p : \Sigma x{:}A.B(x)$, then $\pi_1\, p : A$ and $\pi_2\, p : B(\pi_1\, p)$. So $\pi_1 : \Sigma x{:}A.B(x) \rightarrow A$ and $\pi_2 : \Pi y{:}(\Sigma x{:}A.B(x)).B(\pi_1\, y)$ and there is the usual computation rule for the projections (π_1 and π_2) and pairing ($\langle \_, \_ \rangle$).

4.1.2 Inductive Types

Taking the idea of sets defined via construction principles as basis, a general pattern for defining types by induction emerges. This idea originates from Scott [Sco70] and Martin-Löf [ML84]; the syntax we present below is loosely based on [CP90, PM93] and the formalisation of inductive types in the proof

assistant Coq [Coq04]. Basically, an inductive type X is completely captured by giving its *constructors*, constant terms that have a type of the form

$$A_1 \to A_2 \to \ldots A_n \to X$$

where the type expressions A_i can only contain X in a *strictly positive* position (i.e. A_i does not contain X or is of the form $B_1 \to B_2 \to \ldots B_m \to X$ with X not in B_j). In some applications, the condition of strict positivity may be relaxed to positivity, but in type theories with dependent types one cannot in general.

The constructors are seen as the (only) ways of constructing terms of the type, so one is actually describing the free algebra over the terms generated from these constructors, stated otherwise X is a solution to the domain equation $X = \sigma_1 + \ldots + \sigma_k$, if the σ_i's correspond to the types of the constructors in the following way: if $A_1 \to A_2 \to \ldots A_n \to X$ is the type of the first constructor, then $\sigma_1 = A_1 \times A_2 \times \ldots A_n$. Such a free algebra amounts to two properties:

- Coverage: if $t : X$, then $t = c(s_1, \ldots, s_n)$ for some constructor c.
- No confusion (for terms of type X): $c(t_1, \ldots, t_n) = c'(s_1, \ldots, s_m)$ if and only if $c = c'$, $n = m$ and $t_i = s_i$ for all i.

In type theory with inductive types, these properties for X are automatically generated from the declaration of the constructors for X, and they are automatically enforced. These properties have both a logical and a computational aspect. 'Coverage' is enforced logically by the induction principle and computationally by the principle of well-founded recursion. 'No confusion' is enforced logically by the fact that we can prove a property of elements of X by an (exhaustive) case distinction. It is enforced computationally by the fact that we can define a function over X by cases.

The terms of an inductive type can be seen as trees, with nodes labelled with constructors. If $c : A_1 \to A_2 \to \ldots A_n \to X$, then a node labelled with c has n subtrees that are either expressions of type A_i (if X does not occur in A_i) or a $B_1 \times B_2 \times \ldots \times B_m$-indexed family of trees (if A_i is of the form $B_1 \to B_2 \to \ldots \to B_m \to X$).

We treat some examples to make these rather abstract ideas concrete. The type of trees with labels in A and nodes in B is given by two constructors.

$$\begin{array}{lcl} \mathsf{leaf} & : & A \to \mathsf{Tree} \\ \mathsf{join} & : & B \to \mathsf{Tree} \to \mathsf{Tree} \to \mathsf{Tree} \end{array}$$

The intention is that this defines the free algebra of trees over leaf-type A and node type B. So, we want $\mathsf{leaf}\ x \neq \mathsf{join}\ y\ t_1\ t_2$ for all x, y, t_1, t_2, and we want to be able to define function over Tree by case distinction and recursing over 'smaller trees'. Finally we want to be able to prove properties of elements of Tree by tree-induction. In type theory with inductive types, this is made possible by

allowing to define terms in the following way.

$$
\begin{array}{l}
\mathsf{Fixpoint\ NCnt}(x : \mathsf{Tree}) : \mathsf{nat} := \\
\mathsf{match}\ x\ \mathsf{with} \\
\mid \mathsf{leaf}\ a \Rightarrow 1 \\
\mid \mathsf{join}\ x\ t_1\ t_2 \Rightarrow (\mathsf{NCnt}\ t1) + (\mathsf{NCnt}\ t2) \\
\mathsf{end}.
\end{array}
$$

Here we borrow the syntax from Coq; the above can be read as definition of a recursive function $\mathsf{NCnt}\colon \mathsf{Tree} \to \mathsf{nat}$ where Fixpoint denotes that we are using recursion. In fact the function NCnt is defined by *structural recursion* over the tree type, meaning that in the body of the function definition, NCnt is only called on arguments that are smaller according to the structure of the inductive type. All functions defined by structural recursion are terminating, but it should be noted that structural recursion is a *syntactic* – and thus decidable – criterion for a function to be terminating. The pattern for function-definition by structural recursion can be generated directly from the definition of the inductive type, which makes it possible for computer systems to support this. See [PM93] for how this is done in the proof assistant Coq. Structural recursion is quite powerful, but for some functions there is quite some work to be done to define them. E.g. the gcd function defined in the following way is not structural recursive.

$$
\begin{array}{l}
\mathsf{Fixpoint\ Gcd}(n\ \ m : \mathsf{nat}) : \mathsf{nat} := \\
\mathbf{if}\ n < m\ \mathbf{then}\ \mathsf{Gcd}(n, m-n) \\
\qquad \mathbf{else\ if}\ n = m\ \mathbf{then}\ n \\
\qquad\qquad \mathbf{else}\ \mathsf{Gcd}(n-m, m)
\end{array}
$$

The type nat is defined as an inductive type with constructors $0 : \mathsf{nat}$ and $S : \mathsf{nat} \to \mathsf{nat}$, so a call of Gcd on $m-n$ is not structurally recursive. To establish termination, we would first have to *prove* that the recursive calls of Gcd are only done on *smaller* arguments, according to some well-founded order, and then the function would be defined by recursion over this well-founded order. Note also that the function Gcd is not terminating at all, because on $n = 0$ or $m = 0$ the recursively called argument isn't smaller, so really this function would be partial, of type $\Pi n, m : \mathsf{nat}.(n \neq 0) \to (m \neq 0) \to \mathsf{nat}$. For a solution to the problem of the restrictiveness of structural recursion, see e.g. [BC05].

The induction principle for an inductive type can also be generated from the definition of the inductive type. As a matter of fact, the induction principle and the recursion principle can be seen as instances of the same syntactic schema, but we won't go into this here: see [PM93] for details. The induction principle for Tree is the term Tree_ind with the following type.

$$
\begin{array}{l}
\mathsf{Tree\_ind} : \forall P{:}\mathsf{Tree} \to \mathsf{Prop}. \\
(\forall a{:}A.P(\mathsf{leaf}\ a)) \to (\forall b{:}B.\forall t_1{:}\mathsf{Tree}.P\ t_1 \to \forall t_2{:}\mathsf{Tree}.P\ t_2 \to P(\mathsf{join}\ b\ t_1\ t_2)) \\
\to \forall t{:}\mathsf{Tree}.P\ t
\end{array}
$$

The power of inductive types lies to a large extent in the fact that many mathematical 'objects' can be defined in an inductive (or recursive) way. Defin-

ing them in inductive type theory then gives the added value that the recursion scheme and the induction principle come 'for free'. Examples of mathematical 'objects' that can be defined as inductive types are the following. (1) Logical connectives, like disjunction which has – given two parameters $A, B : \mathsf{Prop}$ – two constructors $\mathsf{left} : A \to A \vee B$ and $\mathsf{right} : B \to A \vee B$, or the existential quantifier which has – given two parameters $A : \mathsf{Set}$ and $P : A \to \mathsf{Prop}$ – one constructor $\mathsf{pair} : \Pi x{:}A.(P\,x) \to \exists\, A\, P$; here we see the use of a *dependently typed* constructor in the definition of an inductive type. (2) Inductively defined relations, like the 'less or equal' on natural numbers $\leq$, which has as constructors

$$\begin{array}{lcl} \mathsf{le\_}n & : & \forall n : \mathsf{nat.le}\ n\ n \\ \mathsf{le\_}S & : & \forall m, n : \mathsf{nat.le}\ n\ m \to \mathsf{le}\ n\ (Sm) \end{array}$$

In the last examples, we see the use of *dependently typed constructors*. This changes the scheme of the type of a constructor that we described in the beginning of this section. Constructors now have a type $\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.X\,t_1 \cdots t_m$, where X may occur in the A_i only in a strictly positive position (i.e. at the end).

Apart from the scheme for inductive types that we describe here, there is also the possibility to introduce one 'generic' well-ordering type, the so called W-type and to define inductive types as instances of this type. The W-type defines a general type of well-founded trees that can be instantiated to specific sets of trees by choosing the branching types in a specific way [NPS90]. Dybjer [Dyb97] shows that the inductive types we describe above can indeed be represented in this way, but then one has to use an *extensional* type theory, i.e. where functions are equal if they have the same graph, which leads to an undecidable typing relation.

4.1.3 Coinductive Types

Coinductive types were added to the type theory in order to make it capable of dealing with infinite objects [Men91, Geu92, MPC86, Hal90, Gim96]. This extension was done by Hagino [Hag87] using the categorical semantics. The idea behind using the categorical semantics is to consider an ambient category for the type theory, and interpret the *weakly* final coalgebras (i.e. final Coalgebras with uniqueness property dropped) of this category as coinductive types. Alternatively Lindström [Lin89] extended Martin-Löf type theory by coinductive types using the non-well-founded set theoretic semantics; while Mendler et al. [MPC86] Martin-Löf [ML90] and Hallnäs [Hal90] tried to directly extend Martin-Löf's constructive type theory by adding extra typing rules for infinite objects. Mendler [Men91] and Geuvers [Geu92] presented a way to encode coinductive types in type theories altogether simpler than Martin-Löf's type theory. Later, Gimenez [Gim96] extended the calculus of inductive construction by a *cofixpoint scheme* that allows for introducing infinite objects.

Coinductive type theories provide a programming framework for algorithms that deal with infinite objects, and therefore are suitable for exact real arithmetic. Especially, since type theories provide a basis for formal verification

tools, formalising an algorithm in type theory paves the way for verification of that algorithm by means of a theorem prover. Therefore a rigorous analysis of correctness of the algorithms becomes possible by stating these algorithms in the language of coinductive type theory. This is made easier if one can devise a coinductive type theory that is specifically suited for working with real numbers. That is to say, for verification of the algorithms of exact arithmetic, one does not necessarily need a general theory of coinductive types and the full power of type theory. In terms of categorical semantics this means that having the (weakly) final coalgebras of polynomial functors should suffice. However one needs the underlying type theory to be strong enough to formalise all the computable real functions.

This brings up the notion of productivity of infinite objects in type theory and functional programming which is similar (in fact dual) to the notion of termination for finite objects [Dij80, Sij89, Coq94]. A function on streams is productive if it can produce arbitrarily large finite approximations in finite time. The example of multiplication by 3 in Section 2.1 is not a productive function. In fact productivity is very similar to the notion of computability (and continuity, and laziness). In TTE it can be related to the *finiteness property* of type two Turing machines [Wei00, § 2.2]. A domain theoretic treatment of productivity for streams can be found in [Sij89] which is expanded and used in the coinductive treatment of lazy exact arithmetic in [Niq04]. For tackling productivity inside the type theory the notion of *guardedness* is studied by type theorists [Coq94, TT00] and is implemented as the basis for the treatment of coinductive types in Coq proof assistant [Gim96].

Guardedness condition is a syntactic criterion that can be used to ensure the productivity much in the same way (in fact in the dual way) as it can be used to ensure the termination of structurally recursive functions: a recursive function with as recursive argument a term with an inductive type is terminating if the argument of the recursive calls is structurally smaller than than the original argument of the function. This structural order is an inherent order that is inherited from the definition of the inductive type of the recursive argument. According to this order applying constructors of the inductive type generates the successors of a term (recall that inductive types are equivalent to the type of general trees). Dually an infinite object (i.e. a term that has a coinductive type) is productive if the calls to itself inside the body of its definition are immediate arguments of constructors of its coinductive type. The above checks are purely syntactical and hence can be automatised; this is exactly what is done in the guardedness checker of Coq proof assistant.

As an example the following definition is a guarded definition for a stream of natural numbers starting from n.

$$\mathsf{nats}\quad n := \mathsf{cons}\quad n\quad (\mathsf{nats}\quad n+1)$$

This is because the sole occurrence of nats in the right hand side is the immediate argument of (i.e. guarded by) cons, the constructor of the coinductive type of streams.

But the same way as the structural recursion is not powerful enough to capture all valid terminating recursive definitions, the guarded-by-constructor approach does not capture the whole class of productive infinite objects. This is because productivity of streams can in general be reduced to the question whether a subset of $\mathbb{N}$ is infinite, which is an undecidable question [Niq04, § 4.7].

Example of the infinite objects that are not guarded and whose productivity is not syntactically detectable are the filter-like functions that are used in functional programming. In order to formalise such infinite objects one option would be to adhere to semantic approaches, e.g. domain-theoretic methods [Niq05] or topological methods [DGM03]. The other option would be to use a very extended setting of coinductive types that includes polymorphic and dependent coinductive types and adapt advanced type-theoretic methods that are used for tackling general (non-structural) recursion. This is the approach taken by Bertot [Ber05] and is implementable in Coq as all the machinery that is necessary (polymorphic and dependent coinductive types) already exist in Coq.

4.2 Program Extraction

Correctness is an important issue in the implementation of computable analysis (which in practice currently mostly amounts to the implementation of exact real arithmetic.)

There are two directions in which one can develop correct programs. In one direction one starts by writing the concrete program and then afterwards tries to establish its correctness, either using informal reasoning, or by annotating the program with invariants and then proving correctness conditions generated from that, or by refining the types used in the program to be more informative, using a programming language that supports dependent typing. In the other direction, one starts very abstractly and then works towards a concrete program. The methods of program refinement and program extraction both fall in this second category.

With program extraction one starts from a *formalisation* of some mathematical theory, a representation of this theory in the computer that has sufficient detail to allow the computer to establish the correctness by proof checking. This formalisation then is automatically transformed into a computer program which implements the constructive content of this formalised theory. This is a direct application of the realisability implementation of constructive logic. Therefore to be able to extract a program from a formalised theory, in general one needs to formalise the theory using constructive logic. However, there also has been some work on extracting programs from classical proofs [BBS02]. Program extraction has been implemented in many systems, like PX [HN87], Nuprl [CAB$^+$86], Coq [Coq04, Let04], Minlog [BBS$^+$98] and Isabelle [NPW02].

Note that to be able to do program extraction in a proof assistant, the logic of the assistant does not need to be constructive: Isabelle/HOL is based on a classical logic, but it supports program extraction [Ber03].

Because the Curry–Howard–de Bruijn isomorphism corresponds in a natural way to a realisability interpretation, program extraction is popular with proof

assistants that implement type theory. In type theory the proofs of a theorem already are lambda terms, which can be seen as functional programs in a simple programming language. Therefore in type theory program extraction is hardly more than transforming one functional language into another functional language. However, because not all computations in these lambda terms are relevant for the final result of the program, a distinction is made between informative and non-informative data-types.Then, when extracting the program, all parts corresponding to non-informative data-types will be removed.

Program extraction is a popular method for establishing the correctness of implementations of computable analysis. Most proof assistants have a formalisation of the theory of real numbers, and an implementation of exact real arithmetic and computable analysis is seen as an easy side product of this.

Program extraction is an attractive method, but it is unclear whether extracted programs will have a competitive performance. For instance the root finding program extracted from a Coq formalisation of the intermediate value theorem turned out to be unusable in practice [CFGW04, CFS03, CFL05]. Apparently if one wants to extract a reasonable program, then one already needs to be aware of the extraction process when writing the formalisation.

4.3 Constructive Analysis

Constructive Analysis has had a major impact on various topics described in this paper. It has its roots in the intuitionistic mathematics of Brouwer [Bro75, Hey56], which already showed the strong connections between computability and topology even before these fields were properly developed. Heyting then defined formal rules for Brouwer's logic. The interpretation of intuitionistic logic now goes by the name BHK, after Brouwer, Heyting and Kolmogorov.

To prove	one needs to
$A \wedge B$	prove A and prove B
$A \vee B$	choose one and prove it
$A \to B$	provide a method transforming a proof of A into a proof of B
$\forall x A$	provide a construction f such that $f(x)$ is a proof of $A(x)$
$\exists x A$	construct t and prove $A(t)$

Table 1: **BHK interpretation**

When a precise theory of computations became available Kleene developed his realisability interpretation to give a formal model for intuitionistic logic. See Section 3.5. Kleene's first interpretation did capture Brouwer's logic nicely, as explained in section 3.5, but did not capture Brouwer's theory of choice sequences. This was solved by Kleene and Vesley [KV65], using functions on Baire space. As we have seen this is the interpretation that also captures TTE.

As is well-known Brouwer contended that all total real functions are continuous [Bro27]. A statement we can now see is provable in many concrete computational interpretations. In 1967 Bishop [BB85] showed that although Brouwer's

continuity principle is an important guideline, one can do without this assumption by just studying the continuous functions and ignoring any other ones, whether they exist or not. In this way Bishop developed major parts of modern analysis. It turns out that Bishop's mathematics is a convenient generalisation of both recursive and intuitionistic mathematics[3]. It can be interpreted in both these computational models described above, see [TvD88b, BR87].

Bishop's model of computation is deliberately vague about the precise notion of computation. It builds on a primitive notion of 'operation'. Martin-Löf theory of types [ML84] can be used as a satisfactory theory of such operations. In fact, the usual way to treat sets in type theory — that is, using types modulo an equivalence relation, called setoids [Hof95] — was motivated by Bishop's work.

Finally we would like to refer to two recent monographs for more information about constructive mathematics [CS05, BVar]. Moreover, this story is not complete without mentioning formal topology [Sam87, FG82]. A proper description would take us to far from exact arithmetic. However, let us just mention that formal topology may be seen as a way to develop topology or domain theory inside type theory [Sam00, SVV96].

5 Conclusion

We have presented some of the problems and solutions of exact real arithmetic varying from concrete implementations, representation and algorithms to various models for real computation. We then put these models in a uniform framework using realisability, opening the door for the use of type theoretic and coalgebraic constructions both in computing and reasoning about these computations. We have indicated that it is often natural to use constructive logic to reason about these computations.

References

[Abe80] O. Aberth. *Computable Analysis*. McGraw-Hill, 1980.

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science (vol. 3): semantic structures*, pages 1–168. Oxford University Press, Oxford, UK, 1994.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[Bau00] Andrej Bauer. *The Realizability Approach to Computable Analysis and Topology*. PhD thesis, Carnegie Mellon University, 2000.

[3] and of classical mathematics, but that's not the issue here.

[Bau05] Andrej Bauer. Realizability as the connection between computable and constructive mathematics. 2005.

[BB85] Erret Bishop and Douglas Bridges. *Constructive Analysis*. Number 279 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, Berlin, 1985.

[BBH01] Jens Blanck, Vasco Brattka, and Peter Hertling, editors. *Computability and Complexity in Analysis: 4th International Workshop, CCA 2000, Swansea, UK, September 17–19, 2000, Selected Papers*, volume 2064 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[BBS+98] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof theory at work: Program development in the Minlog system. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*, Dordrecht, 1998. Kluwer Academic Publishers.

[BBS02] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined Program Extraction from Classical Proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

[BC05] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical. Structures in Comp. Sci.*, 15(4):671–708, 2005.

[BCRO86] Hans-J. Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 162–173. ACM Press, 1986.

[Bee85] Michael J. Beeson. *Foundations of constructive mathematics*. Springer-Verlag, Berlin, Heidelberg, New York, 1985.

[Ber03] Stefan Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[Ber05] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21–23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer-Verlag, 2005.

[BES02] A. Bauer, M.H. Escardó, and A. Simpson. Comparing functional paradigms for exact real-number computation. In *Automata, languages and programming*, volume 2380 of *Lecture Notes in Comput. Sci.*, pages 489–500. Springer, 2002.

[BG01] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In *Handbook of automated reasoning*, pages 1149–1238. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[Bir00] Lars Birkedal. *Developing theories of types and computability via realizability*, volume 34 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., Amsterdam, 2000.

[BM96] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, Stanford, California, 1996.

[Boe05] Hans-J. Boehm. Constructive Reals Calculator. `http://www.hpl.hp.com/personal/Hans_Boehm/crcalc/`, 2005.

[BR87] Douglas Bridges and Fred Richman. *Varieties in Constructive Mathematics*. Number 97 in London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987.

[Bre91] Claude Brezinski. *History of Continued Fractions and Padé Approximants*, volume 12 of *Springer Series in Computational Mathematics*. Springer-Verlag, 1991.

[Bro21] L. E. J. Brouwer. Besitzt jede reelle Zahl eine Dezimalbruchentwicklung? *Math. Ann.*, 83(3-4):201–210, 1921.

[Bro27] L. E. J. Brouwer. Über Definitionsbereiche von- Funktionen. *Math. Ann.*, 97(1):60–75, 1927.

[Bro75] L.E.J. Brouwer. *Collected Works*. North-Holland, 1975.

[Bru80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. R. Hindley and J. P. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 580–606. Academic Press, London, 1980.

[BVar] Douglas Bridges and Luminita Vita. *Techniques of Constructive Analysis*. Springer-Verlag Universitext, to appear.

[CAB+86] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[Ceĭ59] G. S. Ceĭtin. Algorithmic operators in constructive complete separable metric spaces. *Dokl. Akad. Nauk SSSR*, 128:49–52, 1959.

[Ceĭ62] G. S. Ceĭtin. Algorithmic operators in constructive metric spaces. *Trudy Mat. Inst. Steklov.*, 67:295–361, 1962.

[CFGW04] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN: the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management, Proceedings of MKM 2004, Białowieza, Poland*, volume 3119 of *LNCS*, pages 88–103. Springer-Verlag, 2004.

[CFL05] L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. *Electronic Notes in Theoretical Computer Science*, 2005.

[CFS03] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2000*, LNCS, pages 205–220. Springer-Verlag, 2003.

[Coq94] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1994.

[Coq04] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2004. `ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual.ps.gz`.

[CP90] T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88: Proceedings of the international conference on Computer logic*, pages 50–66, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[CS05] Laura Crosilla and Peter Schuster. *From Sets and Types to Topology and Analysis – Towards Practicable Foundations for Constructive Mathematics*. Number 48 in Oxford Logic Guides. Oxford University Press, 2005.

[DEMY02] Zilin Du, Maria Eleftheriou, José E. Moreira, and Chee Yap. Hypergeometric functions in exact geometric computation. In Vasco Brattka, Matthias Schröder, and Klaus Weihrauch, editors, *CCA 2002 Computability and Complexity in Analysis*, volume 66 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2002. Elsevier Science Publishers. 5th International Workshop, CCA 2002, Málaga, Spain, July 12–13, 2002.

[DG96] Pietro Di Gianantonio. A golden ratio notation for the real numbers. Technical Report CS-R9602, Centrum voor Wiskunde en Informatica (CWI), January 1996.

[DGM03] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and

Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24–28, 2002. Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 148–161. Springer-Verlag, 2003.

[Dij80] Edsger W. Dijkstra. On the productivity of recursive definitions. Personal note EWD 749, `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF`, September 1980.

[Dyb97] Peter Dybjer. Representing inductively defined sets by wellorderings in martin-löf's type theory. *Theor. Comput. Sci.*, 176(1-2):329–335, 1997.

[Eda97] Abbas Edalat. Domains for computation in mathematics, physics and exact real arithmetic. *Bull. Symbolic Logic*, 3(4):401–452, 1997.

[Eda05] Abbas Edalat. Exact Computation – Implementations. `http://www.doc.ic.ac.uk/~ae/exact-computation/#bm:implementations`, 2005.

[EE96] A. Edalat and M.H. Escardó. Integration in Real PCF (extended abstract). In *Proceedings of the 11th Annual IEEE Symposium on Logic In Computer Science*, pages 382–393, 1996.

[EH02] Abas Edalat and Reinhold Heckmann. Computing with real numbers: (i) lft approach to real computation, (ii) domain-theoretic model of computational geometry. In Luis Pinto Gilles Barthe, Peter Dybjer and Joao Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 193–267. Springer, 2002.

[EHS04] M.H. Escardó, M. Hofmann, and T. Streicher. On the non-sequential nature of the interval-domain model of real-number computation. *Mathematical Structures in Computer Science*, 14(6):803–814, 2004.

[EL04] Abbas Edalat and André Lieutier. Domain theory and differential calculus (functions of one variable). *Mathematical Structures in Computer Science*, 14(6):771–802, 2004.

[EP97] Abbas Edalat and Peter J. Potts. A new representation for exact real numbers. In Stephen Brookes and Michael Mislove, editors, *Mathematical Foundations of Progamming Semantics, Thirteenth Annual Conference (MFPS XIII), Carnegie Mellon University, Pittsburgh, PA, USA, March 23–26, 1997*, volume 6 of *ENTCS*. Elsevier Science Publishers, 1997.

[EPS99] Abbas Edalat, Peter J. Potts, and Philipp Sünderhauf. Lazy computation with exact real numbers. In Michael Berman and Seth Berman, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP-98), Baltimore, Maryland, USA, September 27-29, 1998*, volume 34(1) of

ACM SIGPLAN Notices, pages 185–194, New York, 1999. ACM Press.

[ES99] M.H. Escardó and Th. Streicher. Induction and recursion on the partial real line with applications to Real PCF. *Theoretical Computer Science*, 210(1):121–157, 1999.

[Esc96] M.H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, 1996.

[FG82] M. P. Fourman and R. J. Grayson. Formal spaces. In *The L. E. J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*, volume 110 of *Stud. Logic Found. Math.*, pages 107–122. North-Holland, Amsterdam, 1982.

[Fil05] Jean-Christophe Filliâtre. Constructive reals OCaml library. `http://www.lri.fr/~filliatr/creal.en.html`, 2005.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proc. of Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. `http://www.cs.kun.nl/~herman/BRABasInf_RecTyp.ps.gz`.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communicants*. PhD thesis PhD 96-11, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1996.

[GL01] Paul Gowland and David Lester. A survey of exact arithmetic implementations. In Blanck et al. [BBH01], pages 30–47.

[GN02] Herman Geuvers and Milad Niqui. Constructive real numbers in Coq: Axioms and categoricity. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs : International Workshop, TYPES 2000, Durham, UK, December 8–12, 2000. Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 79–95. Springer-Verlag, 2002.

[Gon05] Georges Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2005. `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. Elsevier, 1990.

[Guy05] Martin Guy. bignum / BigFloat. http://medialab.freaknet.org/bignum/, 2005.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis CST-47-87, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh, September 1987.

[Hal90] Lars Hallnäs. On the syntax of infinite objects: an extension of Martin-Löf's theory of expressions. In Martin-Löf and Mints [MLM90], pages 94–194.

[Har94] John Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 5:35–59, 1994.

[Hey56] Heyting, A. *Intuitionism. An introduction.* Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, Amsterdam, 1956.

[HLP+05] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann, Sylvie Boldo, David Daney, Mathieu Dutour, Emmanuel Jeandel, Laurent Fousse, Fabrice Rouillier, and Kevin Ryde. The MPFR Library. http://www.mpfr.org/, 2005.

[HN87] S. Hayashi and H. Nakano. PX, a Computational Logic. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[HN05] Jesse Hughes and Milad Niqui. Admissible digit sets. *Theoretical Computer Science, In Press*, 2005. Available online 21 October 2005, http://www.sciencedirect.com/science/article/B6V1G-4HCN79X-8/2/df5d7430%e2c38fdcbcbdadb63fba1e39.

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995. http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-327/.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980. Dedicated to Haskell B. Curry on the occasion of his 80th birthday.

[Hyl82] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. Van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 165–216. North Holland Publishing Company, 1982.

[IEE85] IEEE Task P754. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1985.

[Jac99] Bart Jacobs. *Categorical logic and type theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1999.

[Jac05] Bart Jacobs. Introduction to Coalgebra. Towards Mathematics of States and Observations. Draft available from `http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf`, 22 April 2005.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science. EATCS*, 62:222–259, 1997.

[Kea96] Baker R. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996. available online as `http://interval.louisiana.edu/preprints/survey.pdf`.

[KLS57a] G. Kreisel, D. Lacombe, and J. R. Shoenfield. Partial recursive functionals and effective operations. In *Constructivity in mathematics: Proceedings of the colloquium held at Amsterdam, 1957 (edited by A. Heyting)*, Studies in Logic and the Foundations of Mathematics, pages 290–297, Amsterdam, 1957. North-Holland Publishing Co.

[KLS57b] Georg Kreisel, Daniel Lacombe, and Joseph R. Shoenfield. Fonctionnelles récursivement définissables et fonctionnelles récursives. *C. R. Acad. Sci. Paris*, 245:399–402, 1957.

[KM88] Peter Kornerup and David Matula. An on-line arithmetic unit for bit-pipelined rational arithmetic. *Journal of Parallel and Distributed Computing*, 5:310–330, 1988.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997. xiv+762pp.

[Kon00] Michal Konečný. *Many-Valued Real Functions Computable by Finite Transducers using IFS-Representations*. PhD thesis, School of Computer Science, The University of Birmingham, October 2000.

[Krä97] Walter Krämer. A priori worst-case error bounds for floating-point computations. In Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, editors, *13th IEEE Symposium on Computer Arithmetic: proceedings, July 6–9, 1997, Asilomar, California, USA*, volume 13 of *Symposium on Computer Arithmetic*, pages 64–73, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE Computer Society Press.

[Kre05] Richard Kreckel. CLN - Class Library for Numbers. `http://www.ginac.de/CLN/`, 2005.

[KV65] S.C. Kleene and R.E. Vesley. *The foundations of intuitionistic mathematics, especially in relation to recursive functions.* North-Holland Amsterdam, 1965.

[Lam05] Branimir Lambov. The RealLib Project. `http://www.brics.dk/~barnie/RealLib/`, 2005.

[Les01] David Lester. Effective continued fractions. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 163–172. IEEE Computer Society Press, 2001.

[Les05] David Lester. Exact Arithmetic Implementations. `http://www.cs.man.ac.uk/arch/dlester/exact.html`, 2005.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université de Paris XI Orsay, 2004.

[Lie04] Peter Lietz. *From Constructive Mathematics to Computable Analysis via the Realizability Interpretation.* PhD thesis, Darmstadt University of Technology, 2004.

[Lin89] Ingrid Lindström. A construction of non-well-founded sets within Martin-Löf's type theory. *Journal of Symbolic Logic*, 54(1):57–64, March 1989.

[LS88] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, Cambridge, 1988. Reprint of the 1986 original.

[Luo94] Zhaohui Luo. *Computation and reasoning: a type theory for computer science.* Oxford University Press, Inc., New York, NY, USA, 1994.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51:159–172, 1991.

[Men00] Oskar Mencer. *Rational Arithmetic Units in Computer Systems.* PhD thesis, Stanford University, February 2000.

[MGH+97] M. Monagan, K. Geddes, K. Heal, G. Labahn, and S. Vorkoetter. *Maple V Programming Guide for Release 5.* Springer-Verlag, Berlin/Heidelberg, 1997.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, 1984.

[ML90] Per Martin-Löf. Mathematics of infinity. In Martin-Löf and Mints [MLM90], pages 149–197.

[MLM90] Per Martin-Löf and Grigori Mints, editors. *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[MM94] Valérie Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7, December 1994.

[MPC86] Nax Paul Mendler, Prakash Panangaden, and Robert L. Constable. Infinite objects in type theory. In *Symposium on Logic in Computer Science (LICS '86)*, pages 249–255, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[Mül01] Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In Blanck et al. [BBH01], pages 222–252.

[Mül05] Norbert Müller. iRRAM – Exact Arithmetic in C++. `http://www.informatik.uni-trier.de/iRRAM/`, 2005.

[Niq04] Milad Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud Universiteit Nijmegen, September 2004.

[Niq05] Milad Niqui. Formalising exact arithmetic in type theory. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *New Computational Paradigms: First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8–12, 2005. Proceedings*, volume 3526 of *Lecture Notes in Computer Science*, pages 368–377. Springer-Verlag, 2005.

[NPS90] B. Nordström, K. Peterson, and J. M. Smith. *Programming in Martin-Löf's Type Theory : an introduction*. Oxford Science Publications, 1990.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[O'C05] Russell O'Connor. Few Digits 0.4.0. `http://r6.ca/FewDigits/`, 2005.

[Pat03] Dirk Pattinson. Computable functions on final coalgebras. In *Proc. of 6th Workshop on Coalgebraic Methods in Computer Science, CMCS'03, Warsaw, 5-6 Apr. 2003*, volume 82, 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[PE97] Peter J. Potts and Abbas Edalat. Exact real computer arithmetic. Technical Report DOC 97/9, Department of Computing, Imperial College, March 1997.

[Pfe01] Frank Pfenning. Logical frameworks. In *Handbook of Automated Reasoning*, pages 1063–1147. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Plo77] Gordon D. Plotkin. Lcf considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Proceedings of the 1st TLCA conference*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[Pot98] Peter J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, University of London, Imperial College, July 1998.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.

[Sam87] Giovanni Sambin. Intuitionistic formal spaces - a first communication. In D. Skordev, editor, *Mathematical logic and its Applications*, pages 187–204. Plenum, 1987.

[Sam00] Giovanni Sambin. Formal topology and domains. In *Electronic notes in theoretical computer science, 35 (Remagen-Rolandseck, 1998)*, volume 35 of *Electron. Notes Theor. Comput. Sci.*, page 14 pp. (electronic). Elsevier, Amsterdam, 2000.

[Sch89] Jerry Schwarz. Implementing Infinite Precision Arithmetic. In *Proc. 9th IEEE Symposium on Computer Arithmetic*, pages 10–17. IEEE Computer Society, September 1989.

[Sco70] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics, pages 237–275. Springer-Verlag, 1970.

[Sco72] D. Scott. Lattice theory, data types, and semantics. In R Rustin, editor, *Formal Semantics of Programming Languages*, volume 2 of *Courant Computer Science Symposia*, pages 65–106. Prentice- Hall, 1972.

[Sco76] Dana Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976. Semantics and correctness of programs.

[Sij89] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 11(4):633–649, October 1989.

[Sim98] Alex K. Simpson. Lazy functional algorithms for exact real functionals. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer, 1998.

[SVV96] Giovanni Sambin, Silvio Valentini, and Paolo Virgili. Constructive domain theory as a branch of intuitionistic pointfree topology. *Theoret. Comput. Sci.*, 159(2):319–341, 1996.

[Tro77] A. S. Troelstra. *Choice sequences.* Clarendon Press, Oxford, 1977. A chapter of intuitionistic mathematics, Oxford Logic Guides.

[Tro92] A. S. Troelstra. Comparing the theory of representations and constructive mathematics. In *Computer science logic (Berne, 1991)*, volume 626 of *Lecture Notes in Comput. Sci.*, pages 382–395. Springer, 1992.

[Tro98] A. S. Troelstra. Realizability. In *Handbook of proof theory*, volume 137 of *Stud. Logic Found. Math.*, pages 407–473. North-Holland, Amsterdam, 1998.

[TT00] Alastair Telford and David Turner. Ensuring Termination in ESFP. *Journal of Universal Computer Science*, 6(4):474–488, April 2000.

[TvD88a] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics: An Introduction,vol I*, volume 121 of *Studies in Logic and The Foundation of Mathematics*. North Holland Publishing Co., Amsterdam, 1988.

[TvD88b] A.S. Troelstra and D. van Dalen. *Constructivism in mathematics. An introduction.* Number 123 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

[vO02] Jaap van Oosten. Realizability: a historical essay. *Math. Structures Comput. Sci.*, 12(3):239–263, 2002. Realizability (Trento, 1999).

[Vui90] Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, August 1990.

[Wei97] Klaus Weihrauch. A foundation for computable analysis. In František Plášil and Keith G. Jeffery, editors, *Theory and Practice of Informatics, 24th Seminar on Current Trends in Theory and Practice of Informatics*, volume 1338 of *Lecture Notes in Computer Science*, pages 104–121, Berlin, 1997. Springer-Verlag.

[Wei00] Klaus Weihrauch. *Computable Analysis. An introduction*. Springer-Verlag, Berlin Heidelberg, 2000. 285 pp.

[WK87] Klaus Weihrauch and Christoph Kreitz. Representations of the real numbers and of the open subsets of the set of real numbers. *Annals of Pure and Applied Logic*, 35:247–260, 1987.

[Wol96] S. Wolfram. *The Mathematica book*. Cambridge University Press, Cambridge, 1996.

[YD95] Chee Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du and F.K. Hwang, editors, *Computing in Euclidean Geometry*. World Scientific Press, 2nd edition, 1995.