Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 3–36.
doi:10.1017/S0960129506005834 Printed in the United Kingdom

Constructive analysis, types and exact real numbers

HERMAN GEUVERS, MILAD NIQUI, BAS SPITTERS
and FREEK WIEDIJK

Radboud University Nijmegen, the Netherlands
Email: spitters@cs.ru.nl

Received 16 December 2006; revised 1 May 2006

In this 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, which opens the door to 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 (Krämer 1997). As a first attempt to prevent this problem, one may use interval arithmetic (Kearfott 1996). A different approach to computing with real numbers is *exact real arithmetic*, which provides a precision-driven approach to computation with real numbers (Yap and Dubé 1995). 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 has found its main applications 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*:

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

(Note that the number 3.999 should not be taken to be an approximation of some number from the real world, but should have exactly 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, which are in increasing sophistication:

1 First we can use floating point arithmetic, using the IEEE 754 numbers implemented (IEEE Task P754 1985) 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 implementing interval arithmetic 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^{42} digits will give the correct answer for x_{10^4}.

Note that with this approach it is the numerical analyst who will need to determine the necessary precision, and it will not be done automatically 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 taken by the computer). Also, if the numerical analyst makes a mistake in his error estimates, there will be no warning that the answer will be incorrect. Thus, 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. In this way, it is the computer rather than the human that determines what precision will be needed.

This approach can be implemented using interval arithmetic (using bounds with sufficiently large precision), or 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 uses functions that map desired precisions to intervals. In this 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. It is interesting to ask what applications these programs may have. Since all data in the real world is inherently imprecise, it may be argued that exact real arithmetic offers no essential improvement over floating point computation for real-world applications. However, its usefulness for mathematics seems quite clear.

Returning to the question of the 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. Most 'higher order' approaches, such as Domains, TTE and Markov's CRM, that we will discuss later can be unified in a realisability framework. 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, say, 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. Finally, we present brief conclusions in Section 5.

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 require 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 of 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, 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 Brouwer (1921) 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 above example, we see that the real numbers do not allow an effective way to compare real numbers, since the problem above arises precisely because we cannot decide whether $x < \frac{1}{3}$ or $x \geqslant \frac{1}{3}$. Similarly, one does not have an effective equality test.

With the advent of computers, other representations for real numbers were considered: partly because of this theoretical shortcoming and partly to allow a more efficient and hardware-compatible internal representation. Some of these non-standard representations had been known for centuries, and others were discovered and further developed by computer scientists in the course of the 20th century. Knuth (Knuth 1997, § 4) gives a thorough historical account of various representations for different number systems (integers, rationals and real numbers). We do not intend to mention all the different representation systems, but will focus on some of the main ideas that are theoretically and practically important. Furthermore, although a study of different representations for integers and rational numbers is relevant for approximative computations with real numbers, we will focus purely on the various approaches for representing real numbers since this is our main interest in this paper.

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

In the following subsections we mention the three main classes of representations[†] – related (but different) classifications can be found in Weihrauch and Kreitz (1987), Weihrauch (2000) and Gowland and Lester (2001).

2.1.1. *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 the basis of the first full implementation of constructive real numbers in a proof assistant and has also been used in formalising the proof of the fundamental theorem of algebra (Geuvers and Niqui 2002). Other constructive formalisations in the literature usually use a modulus of convergence (Troelstra and van Dalen 1988b; Bishop and Bridges 1985; Weihrauch and Kreitz 1987).

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 a representation was proposed in Boehm *et al.* (1986) 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 indexed by $\mathbb{Z}$ rather than $\mathbb{N}$) 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 in (*) in Section 1, then in a decimal variant on Boehm's functional representation, three examples of possible representations for this number are

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

[†] Although there are three main classes, there are, in fact, four subsections. This is because although continued fractions can be viewed as an example of the third class of representation, 'Streams of nested intervals', they are sufficiently important to be described in a subsection of their own.

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

2.1.2. *Dedekind cuts* Dedekind cuts are an alternative approach to representing real numbers and 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 for reasoning 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 Weihrauch (2000, page 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 (Weihrauch and Kreitz 1987).

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 the use of such representations include the formalisation of real numbers in the HOL theorem prover (Harrison 1994) and the formalisation of the real numbers that is used in the formalisation of the 4-colour theorem in the Coq proof assistant (Gonthier 2005).

2.1.3. *Streams of nested intervals* The most standard way of representing real numbers is the decimal representation. This is a positional representation that falls within 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 the 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 given in Section 2.1.1. 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 (Niqui 2004, § 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 $\rho : \Phi^\omega \longrightarrow I$ (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, it is usual to choose as the set of digits a finite set of Möbius maps satisfying some property, while in the literature the larger class of d-contractions is also studied (Konečný 2000). *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 a stream representation for real numbers, the Möbius maps with integer coefficients (which are also known as *linear fractional transformations* (LFT) or *homographic maps*) are considered. Taking the Möbius map with integer coefficients to be I-refining (that is, mapping the closed interval I to itself) forms the basis of Edalat and Potts' approach to lazy exact arithmetic (Edalat and Potts 1997; Potts 1998; Edalat *et al.* 1999). Restricting this further by taking $c = 0$, one arrives at representations by a finite set of *affine maps*, a subclass that includes the standard b-ary representations.

As mentioned earlier, the standard representation is not computationally suitable, but this shortcoming is easily fixed by using one of the variants of radix representation. Different 'fixes' include changing the base to a non-integer base (for example, the golden ratio base (Di Gianantonio 1996)) or increasing the set of digits by introducing negative digits (for example, redundant b-ary representation (Edalat and Potts 1997)). Both these workarounds had been used long before the advent of computers. For example, the introduction of negative digits can be traced back to the number system implicit in the work of the 11th century mathematician Tabari, and it was even used in mechanical computing devices in the 19th century. A detailed historical survey can be found in Knuth (1997, pages 205–208).

Our example number x_{10000}, when using a representation with negative digits, has many different representations. In addition 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 for which the set of digits is infinite is a representation based on continued fractions, which we consider as a separate class of representations.

2.1.4. *Continued fractions* For centuries continued fractions have been used to represent rational and real numbers and elementary functions (Brezinski 1991). As a result, some implementations of exact real arithmetic are based on continued fraction representations. Some of these representations 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 to represent 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 (Vuillemin 1990; Ménissier-Morain 1994). One can allow for negative integers (and disallow 0,1 and -1) obtaining the $\mathbb{Z}$-fractions (Vuillemin 1990; Ménissier-Morain 1994). In the $\mathbb{Z}$-fractions representation, the rational numbers are also 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, but for the representation of real numbers, they suffer from the same computational shortcoming as the standard decimal representation. In order to overcome this, some modifications of these representations have been studied and used for implementing exact arithmetic (Vuillemin 1990; Lester 2001).

There are several ways to obtain a representation based on continued fractions that uses infinite streams for rational and irrational numbers alike, and therefore fall under the general heading of stream representations. One way is to use the redundant Euclidean continued fractions (Vuillemin 1990), which can be seen as an instance of Möbius maps (Potts 1998, Section 7.3). Another way would be to use the Stern–Brocot representation, which is again expressible in terms of Möbius maps (Niqui 2004, Section 5.7.1).

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

$$\rho : \Phi^{\leqslant\omega} \longrightarrow I,$$

where $\Phi^{\leqslant\omega}$ 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 prefer 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), there are many more representations of this number.

2.2. *Implementations*

Several approaches to exact real arithmetic have been implemented in various programming languages in recent years. Some of them have been considered for real-world applications, especially in the fields of visualisation and computational geometry (Yap and Dubé 1995; Du *et al.* 2002). There have also been studies to enhance existing hardware architectures to support exact real arithmetic (Kornerup and Matula 1988; Mencer 2000).

There are three groups of implementations of exact real arithmetic. They are all based on the ideas 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 (Wolfram 1996) and Maple (Monagan *et al.* 1997). These are generally very fast at basic computations, but sometimes lack 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 try to be as fast as possible at it. Examples of this are MPFR from LORIA (Hanrot *et al.* 2005), GiNaC/CLN by Kreckel (Kreckel 2005), iRRAM by Müller (Müller 2001; Müller 2005) and RealLib by Lambov (Lambov 2005). Programs from this group are generally implemented in C++.
— The third group are also systems that have been designed for exact real arithmetic, but not for speed. These systems are mostly part of an effort to move toward a provably correct implementation of exact real arithmetic. Examples of systems like this are Cr by Filliâtre (Filliâtre 2005) (an ML reimplementation of CR, a Java system by Boehm (Boehm *et al.* 1986; Schwarz 1989; Boehm 2005)), ERA by Lester (Lester 2005), Few Digits by O'Connor (O'Connor 2005), Bignum by Guy (Guy 2005) and ICReals by Edalat *et al.* (Potts and Edalat 1997; Potts 1998; Edalat 2005). These systems are generally implemented in a functional programming language such as ML or Haskell. Like the systems in the first group, they generally lack the features 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 of 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), that is, 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 a 'domain' is identified with that of a dcpo, but mostly authors reserve the word 'domain' for a specific type of dcpo (with additional structure), depending on the application one has in mind, see Abramsky and Jung (1994) for an overview of 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 (that is, 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 $\mathbb{N}_\perp$, that is, the set

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

The idea of an 'approximation' is important in dcpos: *a approximates b* (or *a* is 'way below' *b*), written $a \ll b$, if, whenever $b \leqslant \sqcup X$, we have $a \leqslant x$ for some $x \in X$ (where, of course, X 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 $f, g \in [\mathbb{N}_\bot \to \mathbb{N}_\bot]$, $f \ll g$ implies that f is defined only on a finite set of elements ($f(x) \neq \bot$ 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 or ω-algebraic dcpo. In Gunter and Scott (1990), the notion of a domain is identified with an 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:

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

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: $I \leqslant J$ if $I \supseteq J$. This domain was first proposed, in a slightly different form, in Scott (1972). The intervals should be understood as approximations of real numbers, so a smaller interval gives more information, $\mathbb{R}$ is the $\bot$-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 $\cap A \neq \varnothing$; the lub of such a directed set A is its intersection: $\sqcup A := \cap A$, which is again a closed non-empty interval. The nice thing about this dcpo is that it generalises 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 Edalat and Heckmann (2002) and Edalat and Lieutier (2004) for a more detailed study.

RealPCF (Escardó 1996) is a programming language with real as a basic data type and was designed in such a way that its denotational semantics would be given by the nested interval semantics. An interesting feature of RealPCF is that it contains a parallel

'if' construct. It is shown in Escardó *et al.* (2004) that this non-sequentiality is an inherent feature of the nested interval domain. See Escardó and Streicher (1999) and Edalat and Escardó (1996) 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. Putting this another way, $[A]$ is the solution to the domain equation $X = A_\perp \times X$ in the category of dcpos (Niqui 2004, Section 4.5).

The TTE model of computable real valued functions (see Section 3.3) uses as its representation of the reals exactly the data-type $[A]$ – under the proviso that it is '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.

As with 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 was demonstrated in Simpson (1998), which uses PCF (Plotkin 1977) as a programming language, and the data type of streams over $\{-1, 0, 1\}$ as the type of the reals. This approach is compared in Bauer *et al.* (2002) with the RealPCF approach, 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 because 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*

In this section we give a brief introduction to Markov's constructive recursive mathematics (CRM). Historically, this seems to be the first concrete model for exact real computations – see Troelstra and van Dalen (1988a), Bridges and Richman (1987), Beeson (1985) and Aberth (1980) for more information. In CRM one represents an object by a certain partial *recursive* function. We can treat most of the representations described in Section 2.1 in this way, but will just consider the Cauchy representation of the real numbers as a concrete example. A real number x is thus represented by a partial recursive function that 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, as are, for example, 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 that is different 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 (Kreisel *et al.* 1957b; Kreisel *et al.* 1957a; Ceĭtin 1962; Ceĭtin 1959).

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

This 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 defined to be 0 for $x < 0$ and 1 for $x \geqslant 0$. This is not a total function in CRM, that is, we cannot recursively decide whether $x < 0$ or $x \geqslant 0$ for each real number x. Having such a test would solve the halting problem.

This model behaves as expected for concrete functions on the real numbers. 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. This function is defined in the internal logic, but externally one can see 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 (Brouwer 1975; Heyting, A. 1956; Troelstra 1977). Kleene and Vesley captured much of his theory using a realisability interpretation. This interpretation was rediscovered by Weihrauch in his TTE, which we will discuss next.

3.3. *TTE*

TTE (Type Two Effectivity) is one of the many schools of computable analysis, most of which are known to be equivalent. TTE is a theory of computability based on Turing machines with infinite input and infinite output (Weihrauch 1997; Weihrauch 2000). 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). Thus, we 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 works directly on the representations of real numbers. In fact the notion of representation plays a central role 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 that rigorously defines whether a representation of real numbers is computationally suitable. In Section 3.5.2 we show how the definition of an admissible representation was already hidden in Brouwer's example.

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 $p : \Phi^\omega \longrightarrow I$ 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*, that is, for every (partial) continuous $r : \Phi^\omega \longrightarrow I$, there is a continuous $f : \Phi^\omega \longrightarrow \Phi^\omega$ 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 that 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 (Weihrauch 1997).

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 Edalat and Potts (1997), the ternary Stern–Brocot representation for $[0, +\infty]$ (Hughes and Niqui 2006) and the binary Golden ratio notation for $[0, 1]$ (Di Gianantonio 1996).

As stated earlier, the TTE notion of computability, which is based on the admissible representations, is equivalent to most other models of computability (Weihrauch 2000, Section 9).

3.4. *Coalgebras*

Coalgebras (also called 'systems' in Rutten (2000) and Barwise and Moss (1996)) 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 Jacobs (2005).

In the category theoretic semantics for computer science, to any functor there 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 $\mathscr{C}$ be a category, and F be an endofunctor on $\mathscr{C}$. An F-*coalgebra* is a pair $\langle Y, y \rangle$ in which Y is an object of $\mathscr{C}$ and $y : Y \longrightarrow \mathrm{F}(Y)$ is a morphism in $\mathscr{C}$. 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 $f : U \longrightarrow V$ such that $v \circ f = \mathrm{F}(f) \circ u$; that is, the following diagram commutes:

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

One can check 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 in 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

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

or, equivalently, if the following diagram commutes:

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

In an arbitrary category, it is not always easy to say whether a final coalgebra for a given functor exists. However, if a final F-coalgebra does exist, 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 (Jacobs and Rutten 1997). This fixed point property is the reason we are interested in a final coalgebra since it means we can 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 be stated roughly as: two infinite processes are equal if they are bisimilar, that is, if the observable parts are equal and the continuation of the two processes (or the subprocesses) are again bisimilar.

As an example, consider the set of *streams*. In lazy exact arithmetic, real numbers are represented by means of lazy infinite sequences of elements of a set, which are 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 $\langle \mathsf{hd}, \mathsf{tl}\rangle : \Phi^\omega \longrightarrow \Phi \times \Phi^\omega$, one can show that the set of streams is indeed a final coalgebra for F (Rutten 2000). The constructor of the final coalgebras of streams is $\mathsf{cons} : \Phi^\omega \longrightarrow \Phi^\omega$, 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: these are the so-called coalgebra maps. This does not capture all functions on real numbers, just those for which we have suitable partial observations, that is, 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 (Pattinson 2003). A more structural solution would be to interpret coinductive types in a realisability model, which we will present in the next section.

3.5. *Realisability*

In this subsection we will describe realisability – see van Oosten (2002) and Troelstra (1998) for general overviews. After a general introduction to realisability, we will briefly describe three realisability interpretations: for recursive analysis, for TTE and for domain theory. In this way we will obtain a nice uniform treatment of the three models previously discussed. Our presentation of realisability models in this section follows, for the most part, the presentations in Birkedal (2000) and Bauer (2000; 2005), which can be consulted along with van Oosten (2002) for historical background.

In order to represent data on a computer, we need to find a code, a realisation, 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 realised via the following commutative diagram:

$$\begin{array}{ccc} R & \xrightarrow{f'} & R \\ {\scriptstyle \Vdash}\downarrow & & \downarrow{\scriptstyle \Vdash} \\ X & \xrightarrow[f]{} & Y \end{array}$$

It is then said that f' tracks f. These representations are connected to the representations discussed earlier. 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 to be convenient to require the realisers to have the structure of a partial combinatory algebra (PCA). A PCA is a structure $(X, \bullet, \mathbf{k}, \mathbf{s})$ that has all the relevant properties of the combinator presentation of recursion theory. In Kleene's original realisability interpretation the realisers are given by 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. But, how do we include a data-type for all, not just the recursive real numbers? Putting this another way, how does one make a realisability model corresponding to, say, TTE? To do this one needs a slightly different picture. The structures are realised by *all* streams, that is, elements of *Baire space*[†], but we only allow *recursive* functions. To solve this, one uses the notion of *relative realisability*, in which 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 Birkedal (2000) and Bauer (2000) for a simplified version that suffices for the present context.

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 the computability of lists of real numbers, trees of reals numbers, streams of real numbers, the positive real numbers, and so on. 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 that is closed under certain type forming constructions – see Section 4.1. Categorical logic and type theory (Lambek and Scott 1988; Jacobs 1999) allow us to define a very strict and structural connection between logic and semantics by the use of the *internal* logic of a (categorical) model. It is customary to speak of 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 earlier we show that we can use this type theory in all these models. This means that we have a notion of computability on, say, streams of positive real numbers in all these models.

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

$$x \underline{\mathbf{r}} P \;:=\; P \wedge x\downarrow \quad \text{for } P \text{ atomic} \tag{1}$$

$$x \underline{\mathbf{r}} (A \wedge B) \;:=\; (\mathbf{p}_0 x \underline{\mathbf{r}} A) \wedge (\mathbf{p}_1 x \underline{\mathbf{r}} B) \tag{2}$$

$$x \underline{\mathbf{r}} (A \to B) \;:=\; \forall y (y \underline{\mathbf{r}} A \to x \bullet y \underline{\mathbf{r}} B) \wedge x\downarrow \tag{3}$$

$$x \underline{\mathbf{r}} \forall y A \;:=\; \forall y (x \bullet y \underline{\mathbf{r}} A) \tag{4}$$

$$x \underline{\mathbf{r}} \exists y A \;:=\; \mathbf{p}_1 x \underline{\mathbf{r}} A[y/\mathbf{p}_0 x]. \tag{5}$$

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 realised, that is, the set of realisers is

[†] We have used Φ^ω to denote Baire space before where we choose the alphabet Φ to be the natural numbers.

inhabited. A similar definition for the realisation of types can be given analogous to the Curry–Howard isomorphism, which we will discuss in Section 4.1.

We will now go on to give the realisability interpretation for the three models presented earlier.

3.5.1. *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 realisers, 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.

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

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 an unbounded search: if we know that an element with a decidable property cannot fail to exist, we can just start searching until we find it. Formally, let P be a decidable predicate. Then

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

As stated earlier 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, which is the constructive variant of König's lemma. In order to remedy this, one introduces choice sequences, a concept that is captured by Kleene and Vesley's realisability model, which we will discuss now.

3.5.2. *TTE* TTE is the model 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 (Kleene and Vesley 1965). Troelstra (Troelstra 1992) seems to have been the first to observe the possibility of using 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, consider a representation of the real numbers. First, there is no absolute pre-given notion of a 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, $c : \mathbb{B} \to \mathbb{R}$, of the real numbers. One defines a representation $r : \mathbb{B} \to \mathbb{R}$ of the real numbers as a surjective map from Baire space to the real numbers. Of course, surjectivity should be interpreted in the internal logic. 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, which is denoted C-C in Troelstra and van Dalen (1988a), 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 an 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 that we used above holds in the internal logic[†].

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 transfer constructive theorems about such spaces directly to TTE; see Lietz (2004) 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 TTE is much like the relation between an advanced programming language 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.

3.5.3. *Domains* The theory of effectively presented continuous domains as used by Edalat and co-workers (Edalat 1997) fits into the model Mod($\mathbb{P}, \mathbb{P}_\sharp$). Here $\mathbb{P}$ denotes Scott's graph model (Scott 1976), which may be seen as the 'universal' countably based T_0 topological space; see Bauer (2000) for details.

3.5.4. *Coinductive types* As mentioned earlier, 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 (Hyland 1982) and is called the effective topos. Thus, when we define our data-types coinductively, we can interpret them directly 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 interpret these streams directly 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

[†] We have been unable to find this simple observation in the literature.

the connections between constructive analysis and computation with the reals. For more details on type theory, see Martin-Löf (1984), Nordström *et al.* (1990), Luo (1994), Barendregt (1992) and Barendregt and Geuvers (2001) – we will not deal with programming language aspects of type theory (see Pierce (2002) for details), nor shall we discuss logical frameworks (see Pfenning (2001) for details). 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 whether a term is of a given type is *decidable*, since 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 : nat, then $S\,x$: 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 saying 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 in Howard (1980) (but the original paper dates back to 1968). Howard also treated the case of proofs by induction over the natural numbers and 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 (De Bruijn 1980). 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 studied, and some of 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, for example, 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 reason for Martin-Löf's interest in the formulas-as-types isomorphism, which he took as the starting point for his intuitionistic theory of types (Martin-Löf 1984), and which he extended to the existential quantifier and inductive types.

Combining the computational and 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) when $B : \mathsf{Prop}$. A definition is read as a reference to a proved lemma (with proof t) when $B : \mathsf{Prop}$.

The correspondence between logic and type theory is so strong that there is an isomorphism between logic (for example, the $\wedge\to$-fragment of propositional logic) and type theory (the system $\lambda\to\times$). 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 (for example, β-reduction in $\lambda\to\times$).

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 $\leqslant$) should be of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{Prop}$. This phenomenon is called *type dependency*: the possibility of forming type expressions that contain term expressions as subterms. Type dependency also implies the formation of the *dependently 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 $\pi_1 : \Sigma x{:}A.B(x) \to A$ and $\pi_2 : \Pi y{:}(\Sigma x{:}A.B(x)).B(\pi_1\, y)$, and there are the usual computation rules 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 originated with Scott (Scott 1970) and Martin-Löf (Martin-Löf 1984); the syntax we present below is loosely based on Coquand and Paulin (1990) and Paulin-Mohring (1993), and the formalisation of inductive types in the proof assistant Coq (Coq Development Team 2004). Basically, an inductive type X is completely captured by giving its *constructors*, constant terms that have a type of the form

$$A_1 \to A_2 \to \dots A_n \to X$$

where the type expressions A_i can only contain X in a *strictly positive* position (that is, A_i does not contain X or is of the form $B_1 \to B_2 \to \dots B_m \to X$ 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 do this 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, in other words, X is a solution to the domain equation $X = \sigma_1 + \dots + \sigma_k$ if the σ_i's correspond to the types of the constructors as follows: if $A_1 \to A_2 \to \dots A_n \to X$ is the type of the first constructor, then $\sigma_1 = A_1 \times A_2 \times \dots A_n$. Such a free algebra amounts to two properties:

— *Coverage*: if $t : X$, then $t = c(s_1, \dots, s_n)$ for some constructor c.

— *No confusion (for terms of type X)*: $c(t_1, \dots, t_n) = c'(s_1, \dots, s_m)$ 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 $c : A_1 \to A_2 \to \dots A_n \to X$, 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 $B_1 \times B_2 \times \dots \times B_m$-indexed family of trees (if A_i is of the form $B_1 \to B_2 \to \dots \to B_m \to X$).

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

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

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 functions over Tree by case distinction and recursion 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 the definition of terms as follows:

```
Fixpoint NCnt(x : Tree) : nat :=
match x with
| leaf a ⇒ 1
| join x t1 t2 ⇒ (NCnt t1) + (NCnt t2)
end.
```

Here we have borrowed the syntax from Coq; the above can be read as the definition of a recursive function NCnt : Tree → nat where Fixpoint denotes the fact that we are using recursion. This function counts the numbers of leaves in a tree. 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 Paulin-Mohring (1993) 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. For example, the gcd function defined as follows is not structural recursive:

```
Fixpoint Gcd(n m : nat) : nat :=
if n < m then Gcd(n, m − n)
    else if n = m then n
        else Gcd(n − m, m)
```

The type nat is defined as an inductive type with constructors 0 : nat and S : nat→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 $\Pi n, m$: nat.$(n \neq 0) \to (m \neq 0) \to$ nat. For a solution to the problem of the restrictiveness of structural recursion, see, for example, Bove and Capretta (2005).

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 will not go into this here – see Paulin-Mohring (1993) for details. The induction principle for Tree is the term Tree_ind with the following type:

```
Tree_ind : ∀P :Tree→Prop.
(∀a:A.P(leaf a)) → (∀b:B.∀t1 :Tree.P t1 → ∀t2:Tree.P t2 → P(join b t1 t2))
→ ∀t:Tree.P t
```

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. Defining 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:

1 *Logical connectives* Like: disjunction, which has, given two parameters A, B : Prop, two constructors left : $A \to A \vee B$ and right : $B \to A \vee B$; or the existential quantifier, which has, given two parameters A : Set and P : $A \to$Prop, one constructor 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 'less than or equal to' on natural numbers $\leqslant$, which has as constructors

$$\mathsf{le\_}n \ : \ \forall n : \mathsf{nat.le}\ n\ n$$
$$\mathsf{le\_}S \ : \ \forall m, n : \mathsf{nat.le}\ n\ m \to \mathsf{le}\ n\ (Sm).$$

In these 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 (that is, at the end).

Apart from the scheme for inductive types that we describe here, there is also the possibility of introducing 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 (Nordström *et al.* 1990). Dybjer (Dybjer 1997) shows that the inductive types we have described above can indeed be represented in this way, but then one has to use an *extensional* type theory, that is, 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 to enable it to deal with infinite objects (Mendler 1991; Geuvers 1992; Mendler *et al.* 1986; Hallnäs 1990; Giménez 1996). This extension was done by Hagino (Hagino 1987) 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 (that is, final coalgebras with the uniqueness property dropped) of this category as coinductive types. Taking a different approach, Lindström (Lindström 1989) extended Martin-Löf type theory by coinductive types using the non-well-founded set theoretic semantics; while Mendler *et al.* (Mendler *et al.* 1986), Martin-Löf (Martin-Löf 1990) and Hallnäs (Hallnäs 1990) tried to extend Martin-Löf's constructive type theory directly by adding extra typing rules for infinite objects. Mendler (Mendler 1991) and Geuvers (Geuvers 1992) presented a way to encode coinductive types in type theories that are altogether simpler than Martin-Löf's type theory. Later, Gimenez (Giménez 1996) extended the calculus of inductive construction by a *cofixpoint scheme* that allows for the introduction of infinite objects.

Coinductive type theories provide a programming framework for algorithms that deal with infinite objects, and therefore are suitable for exact real arithmetic. In particular,

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. In other words, one does not necessarily need a general theory of coinductive types and the full power of type theory in order to verify the algorithms of exact arithmetic. 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 (Dijkstra 1980; Sijtsma 1989; Coquand 1994). 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 (Weihrauch 2000, Section 2.2). A domain theoretic treatment of productivity for streams can be found in Sijtsma (1989), which is expanded and used in the coinductive treatment of lazy exact arithmetic in Niqui (2004). In order to tackle productivity inside the type theory, the notion of *guardedness* is studied by type theorists (Coquand 1994; Telford and Turner 2000) and is implemented as the basis for the treatment of coinductive types in the Coq proof assistant (Giménez 1996).

The guardedness condition is a syntactic criterion, which 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 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 (that is, 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 the Coq proof assistant.

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

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

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

But, in 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 the productivity of streams can in general be reduced to the question of whether a subset of $\mathbb{N}$ is infinite, which is an undecidable question (Niqui 2004, Section 4.7).

An example of infinite objects that are not guarded and whose productivity is not syntactically detectable are the filter-like functions that are used in functional programming. One option for formalising such infinite objects would be to adhere to semantic approaches, for example, domain-theoretic methods (Niqui 2005) or topological methods (Di Gianantonio and Miculan 2003). 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 (Bertot 2005) and is implementable in Coq as all the machinery that is necessary (polymorphic and dependent coinductive types) already exists 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 approaches for developing correct programs. In one approach one starts by writing the concrete program and then 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 approach, 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, or, rather, 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 is then automatically transformed into a computer program that implements the constructive content of this formalised theory. This is a direct application of the realisability implementation of constructive logic. Therefore, in order to extract a program from a formalised theory, in general one needs to formalise the theory using constructive logic. However, there has also been some work on extracting programs from classical proofs (Berger *et al.* 2002). Program extraction has been implemented in many systems, such as PX (Hayashi and Nakano 1987), Nuprl (Constable *et al.* 1986), Coq (Coq Development Team 2004; Letouzey 2004), Minlog (Benl *et al.* 1998) and Isabelle (Nipkow *et al.* 2002).

Note that the logic of a proof assistant does not need to be constructive for it to be able to do program extraction: Isabelle/HOL is based on a classical logic, but supports program extraction (Berghofer 2003).

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 are already lambda terms, which can be seen as functional programs in a simple programming language. Therefore,

Table 1. *BHK interpretation*

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 $A(t)$

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 are 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 (Cruz-Filipe *et al.* 2004; Cruz-Filipe and Spitters 2003; Cruz-Filipe and Letouzey 2005). Apparently, if one wants to extract a reasonable program, one 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 (Brouwer 1975; Heyting, A. 1956), which had already shown 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.

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 3.5.1, but did not capture Brouwer's theory of choice sequences. This was solved by Kleene and Vesley (Kleene and Vesley 1965) 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 (Brouwer 1927). A statement that we can now see is provable in many concrete computational interpretations. In 1967, Bishop (Bishop 1967) 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 others, 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 (and of classical mathematics, but that is not the issue here). It can be interpreted in both of the computational models described above: see Troelstra and van Dalen (1988a) and Bridges and Richman (1987).

Bishop's model of computation is deliberately vague about the precise notion of computation. It builds on a primitive notion of 'operation'. Martin-Löf's theory of types (Martin-Löf 1984) 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 (Hofmann 1995), was motivated by Bishop's work.

Finally, we would like to mention the two recent monographs, Crosilla and Schuster (2005) and Bridges and Vita (to appear), which provide more information about constructive mathematics. Furthermore, this story could not be considered complete without mentioning formal topology (Sambin 1987; Fourman and Grayson 1982): a proper description would take us too far from exact arithmetic, but we should just say that formal topology may be seen as a way to develop topology or domain theory inside type theory (Sambin 2000; Sambin *et al.* 1996).

5. Conclusion

We have described some of the problems of exact real arithmetic and some solutions 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 for both computing and reasoning about these computations. We have also indicated that it is often natural to use constructive logic to reason about these computations.

References

Aberth, O. (1980) *Computable Analysis*, McGraw-Hill.

Abramsky, S. and Jung, A. (1994) Domain theory. In: Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (eds.) *Handbook of logic in computer science (vol. 3): semantic structures*, Oxford University Press 1–168.

Barendregt, H. and Geuvers, H. (2001) Proof-assistants using dependent type systems. In: *Handbook of automated reasoning*, Elsevier Science Publishers 1149–1238.

Barendregt, H. P. (1992) Lambda calculi with types. In: *Handbook of logic in computer science (vol. 2): background, computational structures*, Oxford University Press 117–309.

Barwise, J. and Moss, L. (1996) *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*, CSLI Publications, Stanford, California.

Bauer, A. (2000) *The Realizability Approach to Computable Analysis and Topology*, Ph.D. thesis, Carnegie Mellon University.

Bauer, A. (2005) Realizability as the connection between computable and constructive mathematics. (Available at `http://math.andrej.com/category/papers/`.)

Bauer, A., Escardó, M. and Simpson, A. (2002) Comparing functional paradigms for exact real-number computation. In: Automata, languages and programming. *Springer-Verlag Lecture Notes in Computer Science* **2380** 489–500.

Beeson, M. J. (1985) *Foundations of constructive mathematics*, Springer-Verlag.

Benl, H., Berger, U., Schwichtenberg, H., Seisenberger, M. and Zuber, W. (1998) Proof theory at work: Program development in the Minlog system. In: Bibel, W. and Schmidt, P. H. (eds.) *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*, Kluwer Academic Publishers.

Berger, U., Buchholz, W. and Schwichtenberg, H. (2002) Refined Program Extraction from Classical Proofs. *Annals of Pure and Applied Logic* **114** 3–25.

Berghofer, S. (2003) *Proofs, Programs and Executable Specifications in Higher Order Logic*, Ph.D. thesis, Institut für Informatik, Technische Universität München.

Bertot, Y. (2005) Filters on coinductive streams, an application to eratosthenes' sieve. In: Urzyczyn, P. (ed.) Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005. *Springer-Verlag Lecture Notes in Computer Science* **3461** 102–115.

Birkedal, L. (2000) Developing theories of types and computability via realizability. *Electronic Notes in Theoretical Computer Science* **34**.

Bishop, E. (1967) *Foundations of constructive analysis*, McGraw-Hill.

Bishop, E. and Bridges, D. (1985) *Constructive Analysis*, Grundlehren der mathematischen Wissenschaften **279**, Springer-Verlag.

Boehm, H.-J. (2005) Constructive Reals Calculator. (Available at `http://www.hpl.hp.com/personal/Hans_Boehm/crcalc/`.)

Boehm, H.-J., Cartwright, R., Riggle, M. and O'Donnell, M. J. (1986) Exact real arithmetic: A case study in higher order programming. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*, ACM Press 162–173.

Bove, A. and Capretta, V. (2005) Modelling general recursion in type theory. *Mathematical Structures in Computer Science* **15** (4) 671–708.

Brezinski, C. (1991) *History of Continued Fractions and Padé Approximants*, Springer-Verlag Series in Computational Mathematics **12**.

Bridges, D. and Richman, F. (1987) *Varieties in Constructive Mathematics*, London Mathematical Society Lecture Notes Series **97**, Cambridge University Press.

Bridges, D. and Vita, L. (to appear). *Techniques of Constructive Analysis*, Springer-Verlag Universitext.

Brouwer, L. (1975) *Collected Works*, North-Holland.

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

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

Ceĭtin, G. S. (1959) Algorithmic operators in constructive complete separable metric spaces. *Dokl. Akad. Nauk SSSR* **128** 49–52.

Ceĭtin, G. S. (1962) Algorithmic operators in constructive metric spaces. *Trudy Mat. Inst. Steklov.* **67** 295–361.

Constable, R. L., Allen, S. F., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D. J., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J. T. and Smith, S. F. (1986) *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall.

Coq Development Team (2004) The Coq Proof Assistant Reference Manual. (Available at `ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual.ps.gz`.)

Coquand, T. (1994) Infinite objects in type theory. In: Barendregt, H. and Nipkow, T. (eds.) Types for Proofs and Programs, International Workshop, TYPES'93. *Springer-Verlag Lecture Notes in Computer Science* **806** 62–78.

Coquand, T. and Paulin, C. (1990) Inductively defined types. In: *COLOG-88: Proceedings of the international conference on computer logic*, Springer-Verlag 50–66.

Crosilla, L. and Schuster, P. (2005) *From Sets and Types to Topology and Analysis – Towards Practicable Foundations for Constructive Mathematics*, Oxford Logic Guides **48**, Oxford University Press.

Cruz-Filipe, L., Geuvers, H. and Wiedijk, F. (2004) C-CoRN: the Constructive Coq Repository at Nijmegen. In: Asperti, A., Bancerek, G. and Trybulec, A. (eds.) Mathematical Knowledge Management, Proceedings of MKM 2004. *Springer-Verlag Lecture Notes in Computer Science* **3119** 88–103.

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

Cruz-Filipe, L. and Spitters, B. (2003) Program extraction from large proof developments. In: Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2000. *Springer-Verlag Lecture Notes in Computer Science* 205–220.

De Bruijn, N. G. (1980) A survey of the project AUTOMATH. In: Hindley, J. R. and Seldin, J. P. (eds.) *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press 580–606.

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

Di Gianantonio, P. and Miculan, M. (2003) A unifying approach to recursive and co-recursive definitions. In: Geuvers, H. and Wiedijk, F. (eds.) Types for Proofs and Programs: International Workshop, TYPES 2002. *Springer-Verlag Lecture Notes in Computer Science* **2646** 148–161.

Dijkstra, E. W. (1980) On the productivity of recursive definitions. Personal note EWD 749.

Du, Z., Eleftheriou, M., Moreira, J. E. and Yap, C. (2002) Hypergeometric functions in exact geometric computation. In: Brattka, V., Schröder, M. and Weihrauch, K. (eds.) CCA 2002 Computability and Complexity in Analysis. *Electronic Notes in Theoretical Computer Science* **66**. (Also: 5th International Workshop, CCA 2002, Málaga, Spain, July 12–13 2002.)

Dybjer, P. (1997) Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science* **176** (1–2) 329–335.

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

Edalat, A. (2005) Exact Computation – Implementations. (Available at `http://www.doc.ic.ac.uk/~ae/exact-computation/#bm:implementations`.)

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

Edalat, A. and Heckmann, R. (2002) Computing with real numbers: (i) LFT approach to real computation, (ii) Domain-theoretic model of computational geometry. In: Barthe, G., Dybjer, P., Pinto, L. and Saraiva, J. (eds.) Applied Semantics. *Springer-Verlag Lecture Notes in Computer Science* **2395** 193–267.

Edalat, A. and Lieutier, A. (2004) Domain theory and differential calculus (functions of one variable). *Mathematical Structures in Computer Science* **14** (6) 771–802.

Edalat, A. and Potts, P. J. (1997) A new representation for exact real numbers. In: Brookes, S. and Mislove, M. (eds.) Mathematical Foundations of Progamming Semantics, Thirteenth Annual Conference (MFPS XIII). *Electronic Notes in Theoretical Computer Science* **6**.

Edalat, A., Potts, P. J. and Sünderhauf, P. (1999) Lazy computation with exact real numbers. In: Berman, M. and Berman, S. (eds.) Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP-98). *ACM SIGPLAN Notices* **34** (1) 185–194.

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

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

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

Filliâtre, J.-C. (2005) Constructive reals OCaml library. (Available at `http://www.lri.fr/~filliatr/creal.en.html`.)

Fourman, M.P. and Grayson, R.J. (1982) Formal spaces. In: *The L.E.J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*, Studies in Logic and the Foundations of Mathematics **110**, North-Holland 107–122.

Geuvers, H. (1992) Inductive and coinductive types with iteration and recursion. In: Nordström, B., Pettersson, K. and Plotkin, G. (eds.) Informal Proc. of Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ 193–217.

Geuvers, H. and Niqui, M. (2002) Constructive real numbers in Coq: Axioms and categoricity. In: Callaghan, P., Luo, Z., McKinna, J. and Pollack, R. (eds.) Types for Proofs and Programs: International Workshop, TYPES 2000. *Springer-Verlag Lecture Notes in Computer Science* **2277** 79–95.

Giménez, E. (1996) *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communicants*, Ph.D. thesis, PhD 96-11, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon.

Gonthier, G. (2005) A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge.

Gowland, P. and Lester, D. (2001) A survey of exact arithmetic implementations. In: Blanck, J., Brattka, V. and Hertling, P. (eds.) Computability and Complexity in Analysis: 4th International Workshop, CCA 2000. *Springer-Verlag Lecture Notes in Computer Science* **2064** 30–47.

Gunter, C.A. and Scott, D.S. (1990) Semantic domains. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, Elsevier 633–674.

Guy, M. (2005) bignum/BigFloat. (Available at `http://medialab.freaknet.org/bignum/`.)

Hagino, T. (1987) *A Categorical Programming Language*. Ph.D. thesis CST-47-87, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Hallnäs, L. (1990) On the syntax of infinite objects: an extension of Martin-Löf's theory of expressions. In: Martin-Löf, P. and Mints, G. (eds.) COLOG-88, International Conference on Computer Logic. *Springer-Verlag Lecture Notes in Computer Science* **417** 94–194.

Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P., Boldo, S., Daney, D., Dutour, M., Jeandel, E., Fousse, L., Rouillier, F. and Ryde, K. (2005) The MPFR Library. (Available at `http://www.mpfr.org/`.)

Harrison, J. (1994) Constructing the real numbers in HOL. *Formal Methods in System Design* **5** 35–59.

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

Heyting, A. (1956) *Intuitionism. An introduction*, Studies in Logic and the Foundations of Mathematics, North-Holland.

Hofmann, M. (1995) *Extensional concepts in intensional type theory*, Ph.D. thesis, Laboratory for Foundations of Computer Science, University of Edinburgh. (Available at `http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-327/`.)

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

Hughes, J. and Niqui, M. (2006) Admissible digit sets. *Theoretical Computer Science* **351** (1) 61–73.

Hyland, J. (1982) The effective topos. In: Troelstra, A. and Dalen, D. V. (eds.) *The L.E.J. Brouwer Centenary Symposium*, North-Holland 165–216.

IEEE Task P754 (1985) IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices* **22** (2) 9–25.

Jacobs, B. (1999) *Categorical logic and type theory*, Studies in Logic and the Foundations of Mathematics **141**, North-Holland.

Jacobs, B. (2005) Introduction to Coalgebra. Towards Mathematics of States and Observations. (Draft available at `http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf`.)

Jacobs, B. and Rutten, J. (1997) A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science. EATCS* **62** 222–259.

Kearfott, B. R. (1996) Interval computations: Introduction, uses, and resources. *Euromath Bulletin* **2** (1) 95–112.

Kleene, S. and Vesley, R. (1965) *The foundations of intuitionistic mathematics, especially in relation to recursive functions*, North-Holland.

Knuth, D. E. (1997) *The Art of Computer Programming volume 2: Seminumerical Algorithms*, 3rd edition, Addison-Wesley.

Konečný, M. (2000) *Many-Valued Real Functions Computable by Finite Transducers using IFS-Representations*, Ph.D. thesis, School of Computer Science, The University of Birmingham.

Kornerup, P. and Matula, D. (1988) An on-line arithmetic unit for bit-pipelined rational arithmetic. *Journal of Parallel and Distributed Computing* **5** 310–330.

Krämer, W. (1997) A priori worst-case error bounds for floating-point computations. In: Lang, T., Muller, J.-M. and Takagi, N. (eds.) 13th IEEE Symposium on Computer Arithmetic. *Symposium on Computer Arithmetic* **13** 64–73.

Kreckel, R. (2005) CLN – Class Library for Numbers. (Available at `http://www.ginac.de/CLN/`.)

Kreisel, G., Lacombe, D. and Shoenfield, J. R. (1957a) Fonctionnelles récursivement définissables et fonctionnelles récursives. *C. R. Acad. Sci. Paris* **245** 399–402.

Kreisel, G., Lacombe, D. and Shoenfield, J. R. (1957b) 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, North-Holland 290–297.

Lambek, J. and Scott, P. J. (1988) *Introduction to higher order categorical logic* (reprint of the 1986 original), Cambridge Studies in Advanced Mathematics **7**, Cambridge University Press. .

Lambov, B. (2005) The RealLib Project. (Available at `http://www.brics.dk/~barnie/RealLib/`.)

Lester, D. (2001) Effective continued fractions. In: Burgess, N. and Ciminiera, L. (eds.) *15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press 163–172.

Lester, D. (2005) Exact Arithmetic Implementations. (Available at `http://www.cs.man.ac.uk/arch/dlester/exact.html`.)

Letouzey, P. (2004) *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*, Ph.D. thesis, Université de Paris XI Orsay.

Lietz, P. (2004) *From Constructive Mathematics to Computable Analysis via the Realizability Interpretation*, Ph.D. thesis, Darmstadt University of Technology.

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

Luo, Z. (1994) *Computation and reasoning: a type theory for computer science*, Oxford University Press.

Martin-Löf, P. (1984) *Intuitionistic type theory*, Studies in Proof Theory **1**, Bibliopolis.

Martin-Löf, P. (1990) Mathematics of infinity. In: Martin-Löf, P. and Mints, G. (eds.) COLOG-88, International Conference on Computer Logic. *Springer-Verlag Lecture Notes in Computer Science* **417** 149–197.

Mencer, O. (2000) *Rational Arithmetic Units in Computer Systems*, Ph.D. thesis, Stanford University.

Mendler, N. P. (1991) Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic* **51** 159–172.

Mendler, N. P., Panangaden, P. and Constable, R. L. (1986) Infinite objects in type theory. In: *Symposium on Logic in Computer Science (LICS '86)*, IEEE Computer Society Press 249–255.

Ménissier-Morain, V. (1994) *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*, Thèse, Université Paris 7.

Monagan, M., Geddes, K., Heal, K., Labahn, G. and Vorkoetter, S. (1997) *Maple V Programming Guide for Release 5*, Springer-Verlag.

Müller, N. (2005) iRRAM – Exact Arithmetic in C++. (Available at `http://www.informatik.uni-trier.de/iRRAM/`.)

Müller, N. T. (2001) The iRRAM: Exact arithmetic in C++. In: Blanck, J., Brattka, V. and Hertling, P. (eds.) Computability and Complexity in Analysis: 4th International Workshop, CCA 2000. *Springer-Verlag Lecture Notes in Computer Science* **2064** 222–252.

Nipkow, T., Paulson, L. and Wenzel, M. (2002) Isabelle/HOL — A Proof Assistant for Higher-Order Logic. *Springer-Verlag Lecture Notes in Computer Science* **2283**.

Niqui, M. (2004) *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*, Ph.D. thesis, Radboud Universiteit Nijmegen.

Niqui, M. (2005) Formalising exact arithmetic in type theory. In: Cooper, S. B., Löwe, B. and Torenvliet, L. (eds.) New Computational Paradigms: First Conference on Computability in Europe, CiE 2005. *Springer-Verlag Lecture Notes in Computer Science* **3526** 368–377.

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

O'Connor, R. (2005) Few Digits 0.4.0. (Available at `http://r6.ca/FewDigits/`.)

Pattinson, D. (2003) Computable functions on final coalgebras. In: Proc. of 6th Workshop on Coalgebraic Methods in Computer Science, CMCS'03. *Electronic Notes in Theoretical Computer Science* **82** (1).

Paulin-Mohring, C. (1993) Inductive definitions in the system Coq – rules and properties. In: Bezem, M. and Groote, J. F. (eds.) Proceedings of the 1st TLCA conference. *Springer-Verlag Lecture Notes in Computer Science* **664** 328–345.

Pfenning, F. (2001) Logical frameworks. In: *Handbook of Automated Reasoning*, Elsevier Science Publishers 1063–1147.

Pierce, B. C. (2002) *Types and Programming Languages*, MIT Press.

Plotkin, G. D. (1977) LCF considered as a programming language. *Theoretical Computer Science* **5** (3) 225–255.

Potts, P. J. (1998) *Exact Real Arithmetic using Möbius Transformations*, Ph.D. thesis, University of London, Imperial College.

Potts, P. J. and Edalat, A. (1997) Exact real computer arithmetic. Technical Report DOC 97/9, Department of Computing, Imperial College.

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

Sambin, G. (1987) Intuitionistic formal spaces – a first communication. In: Skordev, D. (ed.) *Mathematical logic and its Applications*, Plenum 187–204.

Sambin, G. (2000) Formal topology and domains. In: Remagen-Rolandseck 1998. *Electronic notes in theoretical computer science* **35**.

Sambin, G., Valentini, S. and Virgili, P. (1996) Constructive domain theory as a branch of intuitionistic pointfree topology. *Theoretical Computer Science* **159** (2) 319–341.

Schwarz, J. (1989) Implementing Infinite Precision Arithmetic. In: *Proc. 9th IEEE Symposium on Computer Arithmetic* 10–17.

Scott, D. (1970) Constructive validity. In: Lacombe, D. and Laudelt, M. (eds.) Symposium on Automatic Demonstration. *Springer-Verlag Lecture Notes in Mathematics* 237–275.

Scott, D. (1972) Lattice theory, data types, and semantics. In: Rustin, R. (ed.) *Formal Semantics of Programming Languages*, Courant Computer Science Symposia **2**, Prentice-Hall 65–106.

Scott, D. (1976) Data types as lattices. *SIAM J. Comput.* **5** (3) 522–587.

Sijtsma, B. A. (1989) On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **11** (4) 633–649.

Simpson, A. K. (1998) Lazy functional algorithms for exact real functionals. In: Brim, L., Gruska, J. and Zlatuska, J. (eds.) Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98. *Springer-Verlag Lecture Notes in Computer Science* **1450** 456–464.

Telford, A. and Turner, D. (2000) Ensuring Termination in ESFP. *Journal of Universal Computer Science* **6** (4) 474–488.

Troelstra, A. and van Dalen, D. (1988a) *Constructivism in mathematics. An introduction*, Studies in Logic and the Foundations of Mathematics **123**, North-Holland.

Troelstra, A. S. (1977) *Choice sequences: A chapter of intuitionistic mathematics*, Oxford Logic Guides, Clarendon Press.

Troelstra, A. S. (1992) Comparing the theory of representations and constructive mathematics. In: Computer science logic (Berne 1991). *Springer-Verlag Lecture Notes in Computer Science* **626** 382–395.

Troelstra, A. S. (1998) Realizability. In: *Handbook of proof theory*, Studies in Logic and the Foundations of Mathematics **137**, North-Holland 407–473.

Troelstra, A. S. and van Dalen, D. (1988b) *Constructivism in Mathematics: An Introduction, vol I*, Studies in Logic and the Foundations of Mathematics **121**, North-Holland.

van Oosten, J. (2002) Realizability: a historical essay. Realizability (Trento 1999). *Mathematical Structures in Computer Science* **12** (3) 239–263.

Vuillemin, J. E. (1990) Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers* **39** (8) 1087–1105.

Weihrauch, K. (1997) A foundation for computable analysis. In: Plášil, F. and Jeffery, K. G. (eds.) Theory and Practice of Informatics, 24th Seminar on Current Trends in Theory and Practice of Informatics. *Springer-Verlag Lecture Notes in Computer Science* **1338** 104–121.

Weihrauch, K. (2000) *Computable Analysis: An introduction*, Springer-Verlag.

Weihrauch, K. and Kreitz, C. (1987) 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.

Wolfram, S. (1996) *The Mathematica book*, Cambridge University Press.

Yap, C. and Dubé, T. (1995) The exact computation paradigm. In: Du, D.-Z. and Hwang, F. (eds.) *Computing in Euclidean Geometry*, 2nd edition, World Scientific Press.