Introduction to

EXACT NUMERICAL COMPUTATION

Notes for a tutorial at ISSAC 2000

Martín Escardó
School of Computer Science
St Andrews University
http://www.dcs.st-and.ac.uk/~mhe/

3rd August 2000

Contents

Chapter 1

Introduction

We introduce real number computation with digital computers, as formalized by Turing machines, from both practical and theoretical perspectives. In the computational models that we consider, real number computations can be performed to any desired degree of accuracy.

An important property of the models is that the programmer doesn't need to be concerned with the syntactical details of the notation systems for real numbers in order to develop mathematically correct programs. As far as the programmer is concerned, real numbers are taken in the usual (classical) mathematical sense. Of course, in any approach to real number computation, there must be an underlying machinery (called the operational semantics) for computing a syntactical representative of the mathematical entity denoted by a program (called the semantical denotation). The perfect match of the operational and the denotational semantics is called computational adequacy. This is what ensures that the programmer can ignore the operational semantics, relying only on algebraic and analytic methods, in order to derive mathematically correct programs.

Two topics, computability and complexity, are not developed in this introductory account—see Pour-el and Richards [48], Weihrauch [64] and Ko [35] among other references—but the fundamental properties of the notion of computability are included. Computability theory is concerned with the classification of mathematical entities as either computable or not computable, and complexity theory is concerned with the classification of computable mathematical entities in degrees of consumption of computational resources—space and time. No doubt, these two topics are not only of fundamental importance but also of interest in their own right. In these notes, however, we wish to emphasize the connections between operational and denotational aspects of real number computation.

Another omission is the well-known BSS model of real number computation of Blum, Shub and Smale [10, 9]. In our view, this is best regarded as a mathematical abstraction of floating-point computation. What we study here, however, is an alternative approach to real number computation proposed by Turing in 1937, with ideas going back to Brouwer in 1920.

With recent technological advances on digital computers, which have dramatically increased their speed and their storage capabilities, the theoretical ideas of Brouwer and Turing have been experimentally put in practice. Many (non-commercial) packages for exact real number computation have been implemented in a variety of programming languages. A competition between systems for exact real computation is being organized by David Lester as part of the fourth workshop on *Computability and Complexity in Analysis* in Swansea, Wales in September—see

`http://www.informatik.fernuni-hagen.de/import/cca/cca2000/`

Overview (2) Before introducing exact numerical computation, we discuss the usual problems that arise in computation with floating-point numbers, considering a particular example. (3) We then arrive at exact numerical computation in a series of attempts, considering the same example as a guide. (4) At this point the notions of operational and denotational semantics are introduced. In the above development, only operational aspects have been discussed. (5) To make the connection with the denotational aspects, we make the set of decimal expansions into a topological space, so that notions of limit and continuity are available. (6) Using this, we prove the fact, first discovered by Brouwer, that infinite decimal expansions are inadequate as a notation system for an operational semantics of real number computation. We then prove the fact, also discovered by Brouwer, that computable functions are continuous. (7) We then briefly discuss several alternative notation systems for real number computation that mathematicians, logicians and computer scientists have proposed over the years. They are all equivalent, in the sense that one can effectively translate between them, and a Church-Turing Thesis for real number computation has been formulated. Moreover, they are characterized, among all possible notation systems, by a topological maximality property. Among these systems, we choose signed-digit binary notation as a paradigmatic example. (8) Each number is denoted by (infinitely) many syntactical representations. We show that it is not possible to effectively pick a canonical one. (9) We then discuss how the numerical order can be read off from the signed-digit binary notation of two numbers. We show that there is an effective normalization procedure for pairs of signed-digit infinite numerals, such that, for normal pairs, the numerical and lexicographical orders coincide. In particular, a normal pair of numerals denote the same number if and only if they are equal. (10) After this preliminary discussion on the effective character of the order relation, we discuss the main difficulty in real number computation, namely that the (in)equality predicates are undecidable. We show that this is not as bad as it may seem at first sight, by exhibiting a general computable definition-by-cases scheme based on inequality tests. (11) We finish by briefly discussing functional approaches for real number computation, putting the above ideas together.

Chapter 2

Floating-point computation

In the usual approach to real number computation, one singles out a large, but finite, set $\mathbb{M} \subseteq \mathbb{R}$ of "machine numbers". These are typically "floating-point numbers". A main problem with this approach is that if x and y are machine numbers, then $x + y$, xy, ... are not necessarily machine numbers. Thus, machine numbers fail to form a field, already by failing to be closed under the field operations.

A solution consists in taking machine numbers close to the mathematical value of the expressions. But is this really a solution? Ignoring overflow problems, we now have a sort of closure under the operations, but the field axioms are not satisfied, because the operations have been modified. Thus, if we use algebra and analysis to construct a numerical program, we don't necessarily get a correct or "approximately correct" program.

Example—the logistic map This is the function $f : [0, 1] \to [0, 1]$ defined by

$$f(x) = ax(1 - x),$$

for some constant a. According to Devaney [18], this was first considered as a model of population growth by the Belgian mathematician Pierre Verhulstby in 1845. For example, a value 0.45 may represent 45% of the maximum population of fish in given lake. Our task is, given x_0, to compute the *orbit*

$$x_0,\ f(x_0),\ f(f(x_0)),\ \ldots,\ f^n(x_0),\ \ldots,$$

which collects the population values of successive generations. We wish to compute an initial segment of the orbit for a given initial population x_0. We choose $a = 4$, as at this value of the constant the logistic map becomes chaotic, in a precise mathematical sense—see e.g. Devaney [18]. For the purposes of our discussion, it suffices to say that its value is sensitive to small variations of its variable.

Let's compute orbits for the same initial value, in simple and double precision. Here is a C program.


```c
#include <stdio.h>
#include <math.h>

void main(void)
{
  float const  a = 4.0;
  float const x0 = 0.671875; /* this is machine representable */
  float        x = x0;
  double       y = x0;
  int i;

  for (i = 0; i <= 60; i++) {
       printf("%d\t%f\t%f\n",i,x,y);

       x = a * x * (1.0-x);
       y = a * y * (1.0-y);
      }
}
```

The last entry of the table produced by the program is

```
60      0.934518        0.928604
```

So $f^{60}(x_0)$ seems to be approximately 0.93. But, is it? Let's see. Here is the full table.

```
0       0.671875        0.671875
1       0.881836        0.881836
2       0.416805        0.416805
3       0.972315        0.972315
4       0.107676        0.107676
5       0.384327        0.384327
6       0.946479        0.946479
7       0.202625        0.202625
8       0.646273        0.646274
9       0.914417        0.914416
10      0.313034        0.313037
11      0.860174        0.860179
12      0.481098        0.481084
13      0.998571        0.998569
14      0.005708        0.005717
15      0.022702        0.022736
16      0.088748        0.088876
17      0.323486        0.323907
18      0.875371        0.875965
19      0.436387        0.434601
20      0.983813        0.982892
```

```
21      0.063698        0.067261
22      0.238564        0.250949
23      0.726604        0.751894
24      0.794603        0.746197 <-- The tables are similar up to here
25      0.652837        0.757549
26      0.906564        0.734675
27      0.338824        0.779711
28      0.896089        0.687047
29      0.372453        0.860054
30      0.934927        0.481445
31      0.243355        0.998623
32      0.736534        0.005501
33      0.776207        0.021884
34      0.694838        0.085621
35      0.848153        0.313159
36      0.515158        0.860362
37      0.999081        0.480558
38      0.003673        0.998488
39      0.014638        0.006039
40      0.057696        0.024009
41      0.217467        0.093730
42      0.680701        0.339779
43      0.869388        0.897317
44      0.454209        0.368558
45      0.991613        0.930892
46      0.033268        0.257328
47      0.128646        0.764442
48      0.448383        0.720282
49      0.989343        0.805903
50      0.042174        0.625693
51      0.161581        0.936805
52      0.541891        0.236806
53      0.992980        0.722916
54      0.027881        0.801234
55      0.108415        0.637033
56      0.386646        0.924888
57      0.948604        0.277882
58      0.195019        0.802655
59      0.627947        0.633600
60      0.934518        0.928604
```

From the table, we can't conclude anything about the value of $f^{60}(x_0)$. Thus, machine-number computation can be

1. ineffective (the answer may not be correct, so we don't get a solution),

2. unreliable (we don't know whether the answer is correct).

This is of course well-known. Numerical analysis tries to predict these problems, and avoid them whenever possible. For example, we learn from numerical analysis that if a square matrix is ill-conditioned then floating-point inversion is unlikely

to produce a result close to mathematical inversion. Some proposed solutions to this problem include the following.

1. Interval arithmetic [40].
   (a) Sometimes effective (the intervals may grow very large),
   (b) always reliable.

2. Stochastic arithmetic [58, 17]. In this approach, results are computed a number of times in floating-point arithmetic, usual two or three, with the intermediate results randomly perturbed. Then probability theory is used to estimate the number of correct digits of the result.
   (a) Sometimes effective,
   (b) probabilistically reliable.

3. Multiple-precision arithmetic (libraries, Mathematica, Maple).
   (a) More effective, but still ineffective,
   (b) more reliable, but still unreliable,
   (c) inefficient.

4. Exact arithmetic, which is the subject of this paper.
   (a) Reliable,
   (b) often effective,
   (c) inefficient—but maybe not as multiple precision,
   (d) sometimes requires different programming techniques.

In principle, multiple-precision arithmetic is as good as exact arithmetic, because for any problem that can be solved using exact arithmetic there is a precision for which multiple-precision computation produces accurate results. There are two problems with this, however. Firstly, it may be difficult, or effectively impossible, to determine the necessary precision in advance. Secondly, the necessary precision may be excessively large, so that *all* intermediate results are computed with that precision when, in practice, only a few of them will actually need that precision to guarantee an accurate overall result.

Chapter 3

Exact numerical computation

In order to solve the problems discussed in the previous chapter, we move from naive attempts to more sophisticated attempts, in which we try to get exact numerical values.

First attempt In the example of the logistic map, we can reliably use *rational arithmetic*, because the logistic map is rational. But we have

1. bad performance, and
2. unfeasible resource consumption.

For $n > 20$, the computation of $f^n(x_0)$ runs out of memory in a standard package for rational arithmetic—the reason is that rather large, relatively prime numerators and denominators arise. But floating-point computation in simple precision already seems to give a fairly accurate answer in this case. Nothing is gained in practice.

Second attempt We can compute the logistic map using *finite*, but *arbitrarily large* binary expansions. Notice that this is different from multiple precision, as the size of a multiple-precision number is fixed in advance. This is

1. reliable, but
2. unfeasible in terms of time and space consumption.

For $n > 10$, the computation of $f^n(x_0)$ takes more than I was patient to wait for. Thus, this is even worse than rational arithmetic. It is easy to see why this is the case. If x has a binary expansion with n digits, then $1 - x$ has often n digits as well. But if x and y have m and n digits then xy has mn digits. Thus, $f^n(x_0)$ has approximately $2^n m$ digits if x_0 has m digits.

Towards a third attempt In functional programming languages such as Haskell, a technique called *lazy evaluation* is used as the underlying computing machinery [6]. The idea is that if one wants to compute a small initial segment of a large sequence, it is not necessary, in general, to evaluate the whole sequence. In fact, in this way one can meaningfully compute with infinite sequences. Here is an example.

```
> from n = n : from (n+1)
> from 0
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ctrl-C
> take 5 (from 11)
  [11, 12, 13, 14,15]
> take 6 (map square (from 1))
  [1, 4, 9, 16, 25, 36]
```

The Haskell operator ":" (pronounced "cons") takes an element x and a sequence s, producing a sequence $x : s$ whose first term is x and whose remaining terms are those of s. For example, $3 : [4, 5, 6] = [3, 4, 5, 6]$. Thus, from a denotational point of view, `from n` is the sequence of all integers starting from n. Operationally, this works more or less as follows. In order to evaluate a list expression e, the computing machinery tries to reduce the expression to the form $[]$ (the empty sequence) or $x : e'$ (a head-normal form). Initially, e' is left unevaluated; if, and only when, more terms of the sequence are required, e' is further reduced (to either the empty sequence or a head-normal form). Thus, if one asks for the expression `from 0` to be evaluated, one gets the infinite sequence of natural numbers. In this case, the execution only stops when one explicitly interrupts it, because only finitely many states are needed to evaluate the expression. For other expressions, infinitely many states may be needed, in which case the execution is interrupted when the computer runs out of memory or the user runs out of patience, whichever happens first. But an infinite list can also be used as an intermediate step in the computation of a finite list. For example, `take 5 (from 11)` takes the first five elements of the infinite list of natural numbers starting from 11, and its computation successfully terminates after finitely many steps producing a finite list.

A slightly more interesting example is this (well-known) implementation of the sieve of Erasthotenes to compute the infinite list of prime numbers:

```
> notdivisibleby m n = (n `mod` m) /= 0
> sieve (p:l) = p : sieve (filter (notdivisibleby p) l)
> prime = sieve (from 2)
> prime
  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,
  79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,
  163,167,173,179, ctrl-C
```

Here the filter function takes a predicate and list, and produces a list containing the elements of the given list that satisfy the predicate.

Third attempt In this attempt we proceed as in the second, but using lazy evaluation. But, as it turns out, if there is an improvement, it is imperceptible. The reason is more or less obvious. In the usual algorithms, which are the ones that we implemented, least significant digits are computed first. Thus, even if one wants only one significative digit of $f^n(x_0)$, more than 2^n digits have to be computed first.

Towards a fourth attempt It is natural to wonder whether there exist algorithms that compute most significant digits first. Let's begin by considering multiplication by three. Assume that we want to multiply a decimal numeral by three, and suppose that, at some stage, the scanned part of the input is "$0 \cdot 3$". The first output digit can be either 0 or 1. If we eventually scan an input digit < 3 then it *has* to be 0, and if we eventually scan a digit > 3 then it *has* to be 1. Thus, while the next scanned digit is 3, we can't determine the first output digit. Therefore, to get the first digit of

$$3 \times 0.3333333333333333333333333333333333333333333$$

we are forced to compute all the 43 digits of the input.

A solution to this problem, first proposed by Cauchy [16] in 1840, is the use of negative digits. For example

$$\begin{aligned} 9\bar{1}5\bar{2} &= 9 \times 10^3 - 1 \times 10^2 + 5 \times 2^1 - 2 \times 2^0 \\ &= 9000 - 100 + 50 - 2 \\ &= 8948, \end{aligned}$$

where we have written the negation symbol on the top of negative digits for clarity. Intuitively, in order to compute a function,

1. we estimate an output digit (after reading some input digits),
2. if it turns out to be not quite correct (after reading more input digits), we can use a negative digit to adjust the estimate.

For example, let's consider multiplication by three again. Suppose that "0.3" is an initial prefix of the input. Then we can safely produce "1." as an initial prefix of the output. In fact, if we later read a digit 0, discovering that the numeral has "0.30" as an initial prefix, we can now output a negative digit $\bar{1}$, having produced "$1.\bar{1}$" up to this point, which is the same as "0.9". A mathematical discussion of these issues is the subject of Chapter 6 below.

Fourth attempt We thus compute with *signed-digit*, arbitrary large binary expansions, processing digits from left to right using a lazy language. As discussed above, computations are performed by demand. For example, if one wants 5 correct digits of $y = h(g(x))$, one may need

1. 49 digits of the intermediate value $g(x)$,
2. 7 digits of the input x.

Six correct decimal digits of $f^{60}(x_0)$ can be computed in a fraction of a second. It is effective up to $f^{250}(x_0)$ in my PC, with a not very careful implementation in the Haskell functional programming language, using an interpreter rather than a compiler.

The following is the output produced by a Haskell implementation. The first column displays six correct decimal digits. Internally, the program first computes $\lceil (6+1)\log_2 10 \rceil = 24$ correct signed binary digits, which are then converted to decimal. For comparison, I have also produced the results with floating-point computations, using the two different, but mathematically equivalent, expressions $f_1(x) = 4x(1-x)$ and $f_2(x) = 1 - (2x-1)^2$ of the logistic map. The floating-point numbers are computed in simple precision and displayed with six digits too. Notice that the floating-point answers differ from those produced by the C implementation, although both languages use the same floating-point standard—I guess that this is due to "optimizations" implemented by the C compiler based on the field properties of real numbers.

```
n       exact          Float 1        Float 2
---------------------------------------------------------------
0       0.671875       0.671875       0.671875
1       0.881836       0.881836       0.881836
2       0.416805       0.416805       0.416805
3       0.972315       0.972315       0.972315
4       0.107676       0.107676       0.107676
5       0.384327       0.384327       0.384327
6       0.946479       0.946479       0.946479
7       0.202625       0.202625       0.202625
8       0.646273       0.646273       0.646273
9       0.914416       0.914417       0.914417
10      0.313037       0.313035       0.313033
11      0.860179       0.860177       0.860174
12      0.481084       0.481091       0.481099
13      0.998569       0.998570       0.998571
14      0.005716       0.005712       0.005707
15      0.022735       0.022720       0.022700
16      0.088875       0.088815       0.088740
17      0.323907       0.323710       0.323462
18      0.875965       0.875688       0.875338
19      0.434601       0.435436       0.436486
20      0.982892       0.983326       0.983864
```

```
21      0.067261      0.065585      0.063503
22      0.250949      0.245135      0.237881
23      0.751894      0.740176      0.725175
24      0.746197      0.769262      0.797184
25      0.757549      0.709991      0.646726
26      0.734675      0.823615      0.913886
27      0.779711      0.581093      0.314793
28      0.687047      0.973695      0.862793
29      0.860054      0.102451      0.473525
30      0.481445      0.367818      0.997196
31      0.998623      0.930112      0.011183
32      0.005501      0.260015      0.044233
33      0.021884      0.769629      0.169108
34      0.085620      0.709201      0.562043
35      0.313159      0.824940      0.984603
36      0.860362      0.577656      0.060641
37      0.480558      0.975878      0.227856
38      0.998488      0.094160      0.703751
39      0.006038      0.341177      0.833942
40      0.024009      0.899100      0.553930
41      0.093730      0.362875      0.988366
42      0.339781      0.924787      0.045993
43      0.897320      0.278223      0.175511
44      0.368548      0.803259      0.578828
45      0.930881      0.632135      0.975145
46      0.257366      0.930161      0.096950
47      0.764515      0.259846      0.350203
48      0.720128      0.769305      0.910244
49      0.806175      0.709899      0.326801
50      0.625028      0.823770      0.880008
51      0.937472      0.580692      0.422376
52      0.234472      0.973955      0.975898
53      0.717980      0.101467      0.094084
54      0.809939      0.364686      0.340930
55      0.615752      0.926760      0.898787
56      0.946406      0.271503      0.363877
57      0.202886      0.791157      0.925882
58      0.646894      0.660911      0.274497
59      0.913689      0.896431      0.796593
60      0.315445      0.371371      0.648129
61      0.863758      0.933818      0.912231
62      0.470720      0.247207      0.320263
63      0.996571      0.744383      0.870779
```

Last attempt Ok, the above is fine, but how about functions such as $1/x$? One could of course truncate the infinite numeral representing a number such as $1/3$ to a finite large numeral, but then the above problems would be reintroduced. In our last, and successful, attempt, we use potentially *infinite* sequences of signed digits, rather than *arbitrary large* sequences. Because our previous algorithms process digit sequences from left to right, it turns out that they already work in

this more general setting.

This is inefficient, compared to floating-point computation, but works. All sorts of improvements (empirically and/or theoretically justified) have been proposed, with good gains in performance, sometimes with more than one order of magnitude, for both time and space. But, despite these advances and unforeseen advances, one is most likely to continue using floating-point approximations and live with round-off errors *when this is possible*.

The point is, however, as the logistic map illustrates, that sometimes floating-point computations don't produce correct answers. In this case, as inefficient as it can be, exact numerical computation is of help. Of course, one can sometimes use symbolic methods, and this is the whole point of computer algebra. But it is often the case that computer algebra gives a symbolic solution to a problem, which then has to be numerically evaluated for some values of the variables. Then, even if the symbolic solution is mathematically correct, one tends to have little confidence on the numerical solution. It is very likely that computer algebra systems will include engines for exact numerical evaluation of symbolic expressions in the near future.

As we mentioned in the introduction, many (non-commercial) packages for exact real number computation have been implemented in a variety of programming languages. In particular, David Plume [46] implemented a framework for exact real number computation as his BSc honour project in the functional programming languages Haskell. His project report, which is available from my home page, is a well-written introductory account to some technical aspects of the subject.

Other approaches are briefly discussed in Chapter 7 below. They consist of the use of more sophisticated notations for real numbers, such as Cauchy sequences of dyadic rationals with fixed rate of convergence, continued fractions, and infinite compositions of Möbius transformations. From the point of view of efficiency, some approaches are better than others for some problems, and worse than others for other problems. However, much of the theory of exact real number computation is independent of the particular notation systems under consideration.

Chapter 4

Operational and denotational semantics

In this chapter we compare the computing machinery of an arbitrary programming language with a built-in data type for real numbers with the mathematical interpretation of its constructs.

Operational semantics As for any programming language, the underlying computing machinery operates on concrete entities. These entities can be sequences of signed binary digits, or any of the notation systems for real numbers discussed in Chapter 7 below. This underlying machinery is called the *operational semantics* of the language. It is this is what makes the language into a *computer* language.

Denotational semantics On the other hand, from the point of view of the programmer, who, in the case of real number computation, is typically a mathematician, a physicist or an engineer, representation details are mostly irrelevant. Whereas the operational semantics assigns computational mechanisms to program constructs, the *denotational semantics* assigns mathematical entities to them. These entities are real numbers, functions, sequences etc.

Example In order to make the distinction between operational and denotational semantics clear, consider for example the command

```
y := 4x(1-x)
```

in a typical programming language. The denotational meaning of this is clear: the state of the store is changed so that the location called y holds the mathematical value $4x(1-x)$ after execution. The operational meaning of this will vary. Under floating-point computation, this will cause some memory locations to be transfered to registers, some operations on the registers to be performed,

and some registers to be transfered to memory. Under exact computation, what actually happens is much more complicated. To begin with, x isn't a memory location anymore; now it is a "real number generator", that is, a piece of code that generates as many digits of a given number as we are patient to wait for. Also, for example, the multiplication operation is now implemented as a procedure that builds a third real number generator from two given generators.

Computational adequacy Of course, one expects the operational and the denotational semantics to match. This property is known as *computational adequacy*. If computational adequacy holds, one can develop correct programs using algebra and analysis, completely ignoring the details of the operational semantics. (Of course, in order to obtain efficient programs, some aspects of the operational semantics will have to be taken into account. But, again, for such purposes, it is often sufficient to consider abstractions of the operational semantics.)

Examples As the basic operations on floating-point numbers don't match the basic operations on real numbers, computational adequacy doesn't hold for floating-point computation—this is another way of putting the well-known fact that computer programs developed using algebra and analysis in a rigorous fashion sometimes produce wrong answers in practice. But computational adequacy has been proved for some data types and programming languages for exact real number computation [25, 47]. This is discussed in Chapter 11 below.

Chapter 5

The space of decimal expansions

In analysis one is interested in approximation processes (limits) and functions that preserve the approximation processes (continuous functions). A topology on a set consists of structure that allows one to define approximation processes in the set. For example, such a structure on the real numbers is induced by the usual notion of distance.

In this chapter we are interested in a topological structure on the set of decimal expansions of real numbers, and its relation to the topological structure on the set of real numbers. Notice the distinction; a decimal expansion is a concrete syntactic entity (a sequence of digits), but a real number is an abstract mathematical entity.

To motivate this, consider a program that runs forever, printing the decimal expansion of π. At any stage of the computation, only a finite prefix of the decimal expansion will be observed. However, a prefix as long as we are patient to wait for will be observed. Moreover, the collection of all such (finite) prefixes uniquely determines the entire (infinite) decimal expansion. Thus, the computation can be thought as a sequence of approximations that converges to the entire decimal expansion.

Now consider a program that runs forever, sometimes reading decimal digits from the input, and sometimes printing decimal digits to the output, so that a function on decimal expansions is implemented. As before, both the input and the output are infinite. However, it is clear that finite amounts of output digits can depend only on finite amounts of input digits. In this sense, the program respects the approximation process describe above and hence is continuous with respect to this notion of approximation.

All this can be made mathematically precise in various ways. For instance, one can define a notion of distance between decimal expansions. For simplicity, we shall consider only fractional infinite decimal expansions in these notes; we therefore ignore the decimal point and the leading digit zero, so that a decimal expansion is just a sequence of decimal digits. We can say, for example, that the distance between two distinct expansions α and β is 2^{-n} where n is the first

position at which they differ—admittedly, the number 2^{-n} is rather arbitrary, and other choices such as $1/n$ work equally well. With this, continuity in the above sense is formalized by the usual $\epsilon - \delta$ definition that occurs in real analysis. In this way, the decimal expansions are made into a metric space.

If we abstract from the metric, whose numerical details are for the most part irrelevant, one gets a topological space. Let $\alpha \equiv_n \beta$ mean that the distance between α and β is smaller than 2^{-n}, that is, that $\alpha_i = \beta_i$ for all $i < n$. Then a set U of decimal expansions is open with respect to this notion of distance iff whenever $\alpha \in U$ there is an n such that $\beta \in U$ for all $\beta \equiv_n \alpha$. A sequence α_i of decimal expansions converges to a decimal expansion β iff for every open set U with $\beta \in U$, as small as we please, there is an n such that $\alpha_i \in U$ for all $i \geq n$. A function ϕ of decimal expansions is continuous with respect to this topology iff for every α and every n there is a k such that $\alpha \equiv_k \beta$ implies $\phi(\alpha) \equiv_n \phi(\beta)$ for all β. The reader who is familiar with topology may have noticed that this way of topologizing decimal expansions is equivalent to endowing the digit set $D = \{0, 1, \ldots, 9\}$ with the discrete topology and then the set $D^{\mathbb{N}}$ of decimal expansions with the product topology.

In the following chapters we use topological arguments to prove computational propositions. Unfortunately, we cannot include all the background details here, so refer the interested reader to Smyth [54] and Vickers [57] for introductions to topology for computer scientists.

Chapter 6

Computational inadequacy of decimal notation

The introduction of decimal notation, centuries ago, was a breakthrough regarding efficient *approximate* computation by hand—and nowadays by electronic digital computers. A perhaps surprising fact, discovered by the constructive mathematician Brouwer in 1920, is that infinite decimal numerals are are not suitable notation for exact computation. Another discovery of Brouwer is that computable functions are continuous (but see Chapter 7 below). We sketch topological proofs these two facts.

The denotation map As above, we only consider fractional numbers, ignoring the decimal point and the leading zero in decimal notation. A *numeral* is an infinite sequence over the digit alphabet $D = \{0, 1, \ldots, 9\}$. A numeral $\alpha \in D^{\mathbb{N}}$ denotes the number

$$[\![\alpha]\!] = \sum_{i \geqslant 0} \alpha_i \cdot 10^{-(i+1)} \in [0, 1].$$

The denotation map $\alpha \mapsto [\![\alpha]\!]$ is the fundamental link between the operational and denotational semantics of real number computation as discussed in Chapter 4—but notice that different operational semantics will use different notation spaces. Our denotation map is a surjection

$$q : D^{\mathbb{N}} \twoheadrightarrow [0, 1].$$

It is not an injection because the decimal rationals $m/10^n \in (0, 1)$ have two decimal notations—the reason is that we consider infinitely long runs of digits 9 as legitimate as infinitely long runs of digits 0, because there is no way of ruling out the former by computational means.

Realizers A function $\phi : D^{\mathbb{N}} \to D^{\mathbb{N}}$ *realizes* a function $f : [0, 1] \to [0, 1]$ if

$$f([\![\alpha]\!]) = [\![\phi(\alpha)]\!],$$

as illustrated in the following diagram:

$$\begin{tikzcd}
D^{\mathbb{N}} \arrow[r, "\phi"] \arrow[d, "q"'] & D^{\mathbb{N}} \arrow[d, "q"] \\
{[0,1]} \arrow[r, "f"'] & {[0,1]}.
\end{tikzcd}$$

Of course, here we are interested in *computable* realizers. The realizer lives at the operational level, whereas the function that it realizes lives at the denotational level. As we discussed above, a necessary (but not sufficient) condition for a function $\phi : D^{\mathbb{N}} \to D^{\mathbb{N}}$ being computable is that finite subsequences of $\phi(\alpha)$ depend only on finite subsequences of α. In this case we say that ϕ is of *finite character*. We state as a proposition a fact that we already discussed in the previous chapter (where D is topologized as in Chapter 5).

Proposition 6.1 *A function $\phi : D^{\mathbb{N}} \to D^{\mathbb{N}}$ is of finite character iff it is continuous.*

This is the first link between computability and continuity, at an *operational level*. To get a link at the denotational level, we state the following.

Proposition 6.2 *The surjection $q : D^{\mathbb{N}} \twoheadrightarrow [0, 1]$ is a topological quotient map.*

Proof Simply because it is continuous, and a continuous surjection of compact Hausdorff spaces is always a quotient map. □

By a basic property of topological quotient maps, we get the following.

Corollary 6.3 *A function $f : [0, 1] \to [0, 1]$ with a realizer $\phi : D^{\mathbb{N}} \to D^{\mathbb{N}}$ of finite character is necessarily continuous.*

Unfortunately, the converse is not true.

Proposition 6.4 *There are continuous functions $f : [0, 1] \to [0, 1]$ with no realizer of finite character, for example $f(x) = 3x/10$.*

Proof Let $\phi : D^{\mathbb{N}} \to D^{\mathbb{N}}$ be a realizer of f. Then, for any α and any $n \geq 0$, the value of $\phi(3^n 2\alpha)$ is of the form 0β. Thus, if ϕ were continuous, $\phi(3^\omega)$ would be of the form $0\beta'$. But, similarly, for any α and any $n \geq 0$, the value of $\phi(3^n 4\alpha)$ is of the form 1β, and continuity of ϕ would imply that $\phi(3^\omega)$ would be of the form $1\beta''$. Thus, $\phi(3^\omega)$ would have to be of the forms $0\beta'$ and $1\beta''$ at the same time. Since this is impossible, we conclude that ϕ cannot be continuous. □

This is of course independent of the choice of a base—but different counterexamples are needed for different bases. The argument of Proposition 6.4 is a topological version of the computational argument already given in Chapter 3.

Chapter 7

Alternative notations

The proposed solutions to this computational problem of decimal notation include the adoption of the following alternative notations for real numbers:

1. Base 2/3 with digits 0 and 1.

   (Brouwer 1920, Turing 1937b)

2. Any integral base with negative digits.

   (Leslie 1817, Cauchy 1840, Avizienis 1964, Wiedmer 1980)

3. Nested sequences of rational intervals.

   (Grzegorczyk 1957)

4. Cauchy sequences of rationals with fixed rate of convergence.

   (Bishop 1967)

5. Continued fractions.

   (Vuillemin 1988)

6. Base Golden Ratio with digits 0 and 1.

   (Di Gianantonio 1996)

7. Infinitely iterated Möbius transformations.

   (Edalat and Potts 1997)

These proposed notations are all equivalent, in the sense that one can effectively translate between them. This is sometimes formulated as a "Church-Turing Thesis" for real number computation. For example, the basic operations, the trigonometric and logarithmic functions, and many functions that occur in analysis are computable with respect to these notation systems.

Topologically, these systems are characterized as maximal in the following sense: given any other notation system for the reals for which the denotation map

is a topological quotient, there is a continuous translation from any of the above notations [36, 63]. For example, decimal notation is not maximal, because it is not possible to continuously translate signed-digit decimal to standard decimal notation—if it were, then it would be possible to continuously multiply by three in decimal notation, which contradicts Proposition 6.4.

Signed-digit notation We take signed-digit binary notation as our paradigmatic example. A *signed-digit numeral* is an infinite sequence over the signed-digit alphabet $\mathbf{3} = \{\bar{1}, 0, 1\}$, where $\bar{1}$ stands for -1. As before, a numeral $\alpha \in \mathbf{3}^{\mathbb{N}}$ denotes the number

$$[\![\alpha]\!] = \sum_{i \geqslant 0} \alpha_i \cdot 2^{-(i+1)} \in [-1, 1],$$

and the surjection $\alpha \mapsto [\![\alpha]\!]$ is a quotient map $q : \mathbf{3}^{\mathbb{N}} \twoheadrightarrow [-1, 1]$. The above definitions and facts for standard numerals apply to signed-digit numerals, except that now one has that, in contrast to Proposition 6.4,

Proposition 7.1 *Every continuous function* $f : [-1, 1] \to [-1, 1]$ *has a realizer* $\phi : \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}}$ *of finite character.*

The same holds for functions of several arguments, with realizers defined in the obvious way.

Proof Müller [41], and Weihrauch and Kreitz [63, 36], showed that the quotient map $q : \mathbf{3}^{\mathbb{N}} \twoheadrightarrow [-1, 1]$ is *admissible* (or maximal) in the following sense. For every quotient map $q' : \mathbf{3}^{\mathbb{N}} \twoheadrightarrow [-1, 1]$, there is a (far from unique) continuous map $t : \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}}$ which translates from q'-notation to q-notation, meaning that $q' = q \circ t$. The same argument shows that, more generally, for every continuous map $g : \mathbf{3}^{\mathbb{N}} \to [-1, 1]$ there is a continuous map $\phi : \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}}$ such that $g = q \circ \phi$. (In other words, the space $\mathbf{3}^{\mathbb{N}}$ is projective over the quotient map q.) Then the result follows by taking $g = f \circ q$. □

Since one can effectively translate between the representations of real numbers discussed in Chapter 7, all of them have the same property.

Computability For the purposes of this paper, a function $f : [-1, 1] \to [-1, 1]$ is computable if it has a computable realizer $\phi : \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}}$. Computability on $\mathbf{3}^{\mathbb{N}}$ can be defined e.g. via Turing machines with (read-only) input tapes and (write-only) output tapes, in additions to the usual (read-write) working tapes [35, 62, 64]. In practice, an intuitive understanding of computability on $\mathbf{3}^{\mathbb{N}}$ is enough for most purposes.

We have seen that computable functions $f : [-1, 1] \to [-1, 1]$ are continuous. More generally, a computable *partial* function is continuous on its domain of definition. For example, the function $1/x$ is continuous and computable on $\mathbb{R} \setminus \{0\}$, but cannot be extended to a continuous function at 0. In practice, one gets non-termination at points of discontinuity.

Chapter 8

Canonical forms

Using the following lemma, it is an easy exercise to see that 0 and 1 have only one signed-digit binary representation, that each binary rational $m/2^n \in (0,1)$ has countably infinite representations, and that every other number has uncountably many representations. Let $\equiv$ be the equivalence relation on numerals induced by the denotation function:

$$\alpha \equiv \beta \text{ iff } [\![\alpha]\!] = [\![\beta]\!].$$

Lemma 8.1 *The following identities hold for all $\alpha \in \mathbf{3}^\star$ and $\beta \in \mathbf{3}^\mathbb{N}$:*

$$\alpha 0 \bar{1} \beta \equiv \alpha \bar{1} 1 \beta, \qquad \alpha 0 1 \beta \equiv \alpha 1 \bar{1} \beta.$$

But each number has two canonically associated representations. Recall that

> $\alpha < \beta$ holds in the lexicographical order iff there is an integer k such that $\alpha_k < \beta_k$ and $\alpha_i = \beta_i$ for each $i < k$.

Proposition 8.2 *Every number is denoted by a smallest and by a largest signed-digit numeral in the lexicographical order.*

Proof The preimage of a point by the quotient map $q : \mathbf{3}^\mathbb{N} \to [-1,1]$ is a closed set because q is continuous. But topologically closed subsets of $\mathbf{3}^\mathbb{N}$ are closed under non-empty infima and suprema in the lexicographical order (in fact, this characterizes topologically closed subsets). □

However, there are no effectively determinable canonical forms:

Proposition 8.3 *There is no continuous, denotation-preserving idempotent map $c : \mathbf{3}^\mathbb{N} \to \mathbf{3}^\mathbb{N}$ such that $c(q^{-1}(\{x\}))$ is a singleton for each $x \in [-1,1]$.*

Proof If there were, $[-1,1]$ would be a retract of $\mathbf{3}^\mathbb{N}$. But $\mathbf{3}^\mathbb{N}$ is a totally disconnected space, and such spaces are closed under retracts. On the other hand, $[-1,1]$ is connected. □

Chapter 9

Numerical order in signed-digit notation

As in the previous chapter, we order numerals lexicographically. If α, β are binary numerals without negative digits, then

$$\alpha \leqslant \beta \text{ implies } [\![\alpha]\!] \leqslant [\![\beta]\!].$$

This property fails for signed-digit numerals (for example, for $\alpha = \bar{1}1^\omega$ and $\beta = 0\bar{1}^\omega$ one has $\alpha < \beta$ but $[\![\alpha]\!] = 0 \nleqslant -1/2 = [\![\beta]\!]$). Moreover, its converse fails for both standard and signed-digit numerals (for example, for $\alpha = 10^\omega$ and $\beta = 01^\omega$ one has that $[\![\alpha]\!] = [\![\beta]\!] = 1/2$ but $\alpha \nleqslant \beta$). However, it turns out that signed-digit numerals admit a very strong order-normalization property that standard numerals don't [28].

Order normalization A pair (α, β) of numerals is *order-normal* it its lexicographical order coincides with its numerical order, in the sense that

1. $\alpha < \beta$ and $[\![\alpha]\!] < [\![\beta]\!]$, or
2. $\alpha = \beta$, or
3. $\alpha > \beta$ and $[\![\alpha]\!] > [\![\beta]\!]$.

Thus, for order-normal pairs (α, β), the condition $[\![\alpha]\!] = [\![\beta]\!]$ implies $\alpha = \beta$. In order to emphasize that order-normality is a property of *pairs* of numerals and not of *single* numerals, we observe that the pair (α, α) is always order-normal.

Theorem 9.1 *There is a computable, denotation-preserving idempotent map*

$$\mathrm{norm} : \mathbf{3}^{\mathbb{N}} \times \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}} \times \mathbf{3}^{\mathbb{N}}$$

whose fixed-points are order-normal pairs.

That is, $\mathrm{norm}(\alpha, \beta) = (\alpha', \beta')$ implies that

1. $([\![\alpha]\!], [\![\beta]\!]) = ([\![\alpha']\!], [\![\beta']\!])$,
2. (α', β') is order-normal, and
3. $\mathrm{norm}(\alpha', \beta') = (\alpha', \beta')$.

In particular, one can always assume w.l.o.g. that any two numerals that denote the same numbers are themselves the same, despite the fact that single numerals don't have effectively determinable canonical forms. Since one can effectively translate between the notation systems of real numbers discussed in Chapter 7, all of them have the same property.

Lemma 8.1 is the fundamental ingredient of the proof of the theorem, which is not included here. Together with topology, the identities of the lemma actually capture all identities induced by the denotation function. Let $\sim$ be the least equivalence relation such that

$$\alpha 0\bar{1}\beta \sim \alpha\bar{1}1\beta, \qquad \alpha 01\beta \sim \alpha 1\bar{1}\beta.$$

The relation $\sim$ is strictly weaker than the relation $\equiv$. For example, $1\bar{1}^\omega \equiv \bar{1}1^\omega$, because both numerals denote the number 0, but they are not related by $\sim$, because it is not possible to obtain one from the other by finitely many applications of the above identities. But if we add limits to $\sim$ then we get $\equiv$.

Proposition 9.2 *The relation $\equiv$ is the topological closure of the relation $\sim$ in the product space $\mathbf{3}^{\mathbb{N}} \times \mathbf{3}^{\mathbb{N}}$.*

(Notice that $\equiv$ has to be closed because the denotation quotient map is continuous and $D^{\mathbb{N}}$ and $[-1, 1]$ are Hausdorff spaces.)

Chapter 10

Inequality tests in real number computation

In all effective approaches to exact real number computation via concrete representations, the (in)equality relations are undecidable. This is not surprising, because an infinite amount of information must be checked in order to decide that two given numbers are equal. In particular, this means that it is not possible to obtain exact algorithms from finite-precision algorithms by simply changing the underlying representation of numbers. The reason is that (in)equality tests are the basic ingredient for branching and looping.

Nevertheless, many definitions by cases consisting of inequalities, such as

$$\min(x, y) = \begin{cases} x & \text{if } x \leqslant y, \\ y & \text{otherwise,} \end{cases}$$

do produce computable functions—but they don't produce algorithms, because such functions cannot be computed by first evaluating the condition and then computing the corresponding branch. As an application of the order-normalization Theorem 9.1, we obtain.

Theorem 10.1 *There is a computable partial function*

$$\text{cases} : [-1, 1]^4 \rightharpoonup [-1, 1]$$

such that

$$\text{cases}(x, t, y, z) = \begin{cases} y & \text{if } x < t \text{ or } y = z, \\ z & \text{if } x > t \text{ or } y = z, \end{cases}$$

with domain of definition $\{(x, t, y, z) \mid x = t \text{ implies } y = z\}$.

Notice that, given this domain of definition, an equivalent specification is

$$\text{cases}(x, t, y, z) = \begin{cases} y & \text{if } x \leqslant t, \\ z & \text{if } x \geqslant t. \end{cases}$$

Proof We first consider an analogous lexicographical case-analysis operator lexcases : $\left(\mathbf{3}^{\mathbb{N}}\right)^4 \rightharpoonup \mathbf{3}^{\mathbb{N}}$ such that

$$\text{lexcases}(\alpha, \beta, \gamma, \delta) = \begin{cases} \gamma & \text{if } \alpha < \beta \text{ or } \gamma = \delta, \\ \delta & \text{if } \alpha > \beta \text{ or } \gamma = \delta, \end{cases}$$

with domain of definition $\{(\alpha, \beta, \gamma, \delta) \mid \alpha = \beta \text{ implies } \gamma = \delta\}$. In order to compute lexcases$(\alpha, \beta, \gamma, \delta)$, we can first output the greatest common prefix of γ and δ. If it is finite then $\gamma \neq \delta$ and hence we must have $\alpha < \beta$ or $\alpha > \beta$. In the first case we output the remainder of γ and in the second the remainder of δ. Now, a realizer for the numerical case-analysis operator is given by $\phi(\alpha, \beta, \gamma, \delta) =$ lexcases$(\alpha', \beta', \gamma', \delta')$ where $(\alpha', \beta') = \text{norm}(\alpha, \beta)$ and $(\gamma', \delta') = \text{norm}(\gamma, \delta)$. □

Although the operator is partial, it can be used to define total functions. For example,

$$\min(x, y) = \text{cases}(x, y, x, y),$$

which shows that min is indeed computable. If $f, g : \mathbb{R} \to \mathbb{R}$ are computable functions that agree at a computable number x_0, then the function $h : \mathbb{R} \to \mathbb{R}$ defined by

$$h(x) = \begin{cases} f(x) & \text{if } x \leqslant x_0, \\ g(x) & \text{if } x \geqslant x_0 \end{cases}$$

is also computable, because

$$h(x) = \text{cases}(x, x_0, f(x), g(x)).$$

Also, some partial functions such as

$$\text{sgn}(x) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{if } x > 0 \end{cases}$$

can be defined by

$$\text{sgn}(x) = \text{cases}(x, 0, -1, 1)$$

and hence are computable.

Notice that the case-analysis operator is a continuous map, defined on a subset of $[-1, 1]^4$, which cannot be extended to a continuous map on any larger subset. In other words, the points (x, t, y, z) with $x = t$ but $y \neq z$ are singularities of the case-analysis operator. This means that the partial character of the case-analysis operator is due to topological rather than recursion-theoretic reasons.

Chapter 11

Functional approaches

The distinction between operational and denotational aspects of built-in data types discussed in Chapter 4 also applies to data types implemented by the programmer, of course. The operational aspects of an *abstract* data type are hidden in their internal implementation, whereas its denotational aspects are part of its specification, which assigns mathematical meaning to the operations available in the external interface.

Turing-complete real number data types By a *Turing-complete* data type it is meant a data type for which all computable operations are definable from the operations available in the interface. Boehm and Cartwright [12] argue that there cannot be any Turing-complete abstract data type for real numbers. Accepting the fact that the computable real numbers are enumerable but not recursively enumerable, by an argument which is essentially the same as Cantor's argument of uncountability of the real numbers, it is easy to see why this is the case.

Partial real numbers and Turing-completeness The same limitations arise in computation over the natural numbers; as it is well-known, the computable total functions are not recursively enumerable, so there cannot be any Turing-complete programming language that defines only total functions [50]. Kleene's idea was to generalize the notion of computable function to partial functions in order to obtain Turing-completeness results—the computable partial functions include the computable functions as a subset, but they are recursively enumerable. Similarly, it is possible add partial real numbers in order to obtain Turing-completeness results for abstract or built-in real number data types. A general theory of partiality in computation, known as *domain theory*, has been developed since the late 60's, initially by Dana Scott, and shortly after by many other people—see e.g. [1, 2, 31, 39, 45]. In this way, Turing-completeness for abstract or built-in data types for real number computation is possible. This has been developed only in the context of higher-order functional programming languages—as

far as we know, no Turing-completeness theorems for imperative data types, in the presence of partial real numbers, has been proved or disproved.

Real PCF PCF is a paradigmatic functional programming language used for theoretical investigations [51] in programming-language semantics. It has operational and denotational semantics connected by an adequacy property [44]. Several extensions of PCF with data types for real numbers have been discussed in the literature [19, 25, 47], and practical versions of these theoretical languages have been implemented by Edalat's group at Imperial College.

The idea is that one starts from a ground type R for real numbers, in addition to the usual ground types N and B for natural numbers and booleans, and that new types are obtained by iterating a function space construction. Thus, the types include

1. functions of real numbers, $R \to R$,
2. predicates defined on real numbers, $R \to B$,
3. sequences of real numbers, $N \to R$,
4. sequences of sequences of real numbers, $N \to (N \to R)$,
5. sequences of functions, $N \to (R \to R)$,
6. functionals mapping sequences to numbers (such as limit operators), $(N \to R) \to R$,
7. functionals mapping functions to numbers (such as definite integration and supremum operators and distributions), $(R \to R) \to R$,
8. functionals mapping predicates to truth-values (such as quantification operators), $(R \to B) \to B$.

and so on. The syntactic framework is a simply typed lambda calculus with recursion [4]—in practice, this means that PCF is a functional programming language such as Haskell or ML [6, 43], limited to the bare minimum level of sophistication, in order to simplify the theoretical investigations. The semantic framework is domain theory, already mentioned above, and one uses an interval domain proposed by Dana Scott as the interpretation of real numbers. The idea is that singletons give complete information or "total real numbers" and that other intervals give incomplete information or "partial real numbers". This is related to interval analysis, of course. But a term of the language denoting the number $\sqrt{2}$ doesn't produce a floating-point interval approximation of $\sqrt{2}$ under the operational semantics; rather, it produces better and better approximations of this value. In practice, this can be a signed-digit binary expansion of $\sqrt{2}$, for example, and, for theoretical purposes, this is taken as a shrinking sequence of rational intervals whose intersection is the singleton interval $\{\sqrt{2}\}$.

Operational semantics of Real PCF The paper [25] contains a development of an operational semantics for PCF extended with real numbers (referred to as Real PCF) and a proof of its computational adequacy, which states that the operational semantics produces a convergent sequence of better and better approximations to the mathematical value of a program. The main idea for developing an operational semantics is the observation that the prefix order of the monoid of increasing affine endomaps of the unit interval is isomorphic to the information (=reverse inclusion) order of the interval domain. Moreover, under the isomorphism, infinitely iterated concatenations in this monoid correspond to joins in the domain. Thus, computations are operationally implemented as infinitely iterated compositions of rational affine maps. These infinitely iterated compositions can be regarded as multi-base expansions allowing not only negative but also rational digits.

In the paper [47] it is proposed to work with the larger monoid of Möbius transformations. This is possible because the information refinement property still holds. Again, a computationally adequate operational semantics is obtained. Advantages of generalizing affine maps to Möbius transformations include the fact that the basic transcendental functions can be elegantly implemented in this framework via the theory of continued fractions, as Edalat and Potts have shown [24].

Turing-completeness of Real PCF A Turing-completeness result for Real PCF is proved in [26]. It says that every computable mathematical entity in the universe of discourse of Real PCF is denoted by at least one program. Thus, one can be reassured that no relevant construction has been unadvertly omitted from the language. Of course, for efficiency or simplicity reasons, one can introduce more constructions, as it is done in [47].

An invariance result was also proved. Essentially, it shows that one hasn't made any mistake in the choice of the effective presentation of the interval domain. It was previously proved by Kanda and Park in 1980 that in general it is possible to effectively present the same domain in different ways that give different sets of computable elements [53, 32]. One thus wonders whether the standard presentation of the interval domain, namely Cantor's enumeration of the rational intervals, is a good choice, or whether there can be cleverer choices that give more computable elements. Let's say that an effective presentation is *reasonable* if it is makes the four basic operations computable and the inequality relation semidecidable. Cantor's presentation is certainly reasonable in this sense. But can we say more? It is proved that any two reasonable effective presentations are equivalent, in the sense that one can effectively translate between them.

Bibliography

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, Oxford, 1994.

[2] R.M. Amadio and P.-L. Currien. *Domains and Lambda-Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[3] A. Avizienis. Binary-compatible signed-digit arithmetic. In *AFIPS Conference Proceedings*, volume 1 of *26*, pages 663–672, 1964.

[4] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

[5] M.J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, New York, 1985.

[6] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, New York, 1988.

[7] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967.

[8] E. Bishop and D. Bridges. *Constructive Analysis*. Springer-Verlag, Berlin, 1985.

[9] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer-Verlag, New York, 1998.

[10] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers. *Bull. Amer. Math. Soc.*, 21:1–46, 1989.

[11] H.J. Boehm. Constructive real interpretation of numerical programs. *SIGPLAN Notices*, 22(7):214–221, 1987.

[12] H.J. Boehm and R. Cartwright. Exact real arithmetic: Formulating real numbers as functions. In Turner. D., editor, *Research Topics in Functional Programming*, pages 43–64. Addison-Wesley, 1990.

[13] H.J. Boehm, R. Cartwright, M. Riggle, and M.J. O'Donnel. Exact real arithmetic: A case study in higher order programming. In *ACM Symposium on Lisp and Functional Programming*, 1986.

[14] V. Brattka. Recursive characterization of computable real-valued functions and relations. *Theoretical Computer Science*, 162(1):45–77, August 1996.

[15] L.E.J. Brouwer. Besitzt jede reelle Zahl eine Dezimalbruchentwicklung? *Math Ann*, 83:201–210, 1920.

[16] A. Cauchy. Sur les moyens d'éviter les erreurs dans les calculs numériques. *Comptes Rendus 11*, pages 789–798, Paris 1840. Republished in: Augustin Cauchy, Œvres complètes, 1ére série, Tome V, pp 431–442.

[17] J.-M. Chesneaux and J. Vignes. Les fondements de l'arithmétique stochastique. *C. R. Acad. Sci. Paris Sér. I Math.*, 315(13):1435–1440, 1992.

[18] R. L. Devaney. *An Introduction to Chaotical Dynamical Systems*. Addison-Wesley, California, 2nd edition, 1989.

[19] P. Di-Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Università Degli Studi di Pisa, Dipartamento di Informatica, 1993.

[20] P. Di-Gianantonio. A golden ratio notation for the real numbers. Technical Report CS-R9602, CWI Amsterdam, 1996.

[21] P. Di-Gianantonio. Real number computability and domain theory. *Information and Computation*, 127(1):11–25, 1996.

[22] A. Edalat. Domains for computation in mathematics, physics and exact real arithmetic. *Bulletin of Symbolic Logic*, 3(4):401–452, 1997.

[23] A. Edalat and M.H. Escardó. Integration in Real PCF. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic In Computer Science*, pages 382–393, New Brunswick, New Jersey, USA, 1996.

[24] A. Edalat and P.J. Potts. A new representation for exact real numbers. In *Mathematical foundations of programming semantics (Pittsburgh, PA, 1997)*, page 14 pp. (electronic). Elsevier, Amsterdam, 1997.

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

[26] M.H. Escardó. Real PCF extended with $\exists$ is universal. In A. Edalat, S. Jourdan, and G. McCusker, editors, *Advances in Theory and Formal Methods of Computing: Proceedings of the Third Imperial College Workshop, April 1996*, pages 13–24, Christ Church, Oxford, 1996. IC Press.

[27] M.H. Escardó. PCF extended with real numbers: A domain-theoretic approach to higher-order exact real number computation. Technical Report ECS-LFCS-97-374, Department of Computer Science, University of Edinburgh, December 1997. PhD thesis at Imperial College of the University of London, 1996. `http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/97/ECS-LFCS-97-374/index.html`.

[28] M.H. Escardó. Effective and sequential definition by cases on the reals via infinite signed-digit numerals. In *Third Workshop on Computation and Approximation (Comprox III)*, volume 13 of *Electronic Notes in Theoretical Computer Science*, 1998. `http://www.elsevier.nl/locate/entcs/`.

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

[30] A. Grzegorczyk. On the definition of computable real continuous functions. *Fund. Math.*, 44:61–77, 1957.

[31] C. A. Gunter. *Semantics of Programming Languages—Structures and Techniques*. The MIT Press, London, 1992.

[32] A. Kanda and D. Park. When are two effectively given domains identical? In K. Weihrauch, editor, *Theoretical Computer Science* 4^{th} *GI Conference*, LNCS, 1979.

[33] S.C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[34] S.C. Kleene and R.E Vesley. *The Foundations of Intuitionistic Mathematics: Especially in Relation to Recursive Functions*. North-Holland, Amsterdam, 1965.

[35] Ker-I Ko. *Complexitity Theory of Real Functions*. Birkhauser, Boston, 1991.

[36] C. Kreitz and K. Weihrauch. Theory of representations. *Theoretical Computer Science*, 38(1):17–33, 1985.

[37] J.D. Lawson. The versatile continuous order. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming*

Languages, volume 298 of *Lecture Notes in Computer Science*, pages 134–160, 1987.

[38] J. Leslie. *The Philosophy of Arithmetic*. Edinburgh, 1817.

[39] M. Mislove. Topology, domain theory and theoretical computer science. *Topology and its Applications*, 89(1–2):3–59, 1998.

[40] R.E. Moore. *Interval analysis*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1966.

[41] N.Th. Müller. Subpolynomial complexity classes of real functions and real numbers. In Laurent Kott, editor, *Proceedings of the 13th International Colloquium on Automata, Languages, and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 284–293, Berlin, 1986. Springer.

[42] J. Myhill. Criteria of constructivity of real numbers. *J. Symbolic Logic*, 18:7–10, 1953.

[43] L.C. Paulson. *ML for the working programmer*. Cambridge University Press, Cambridge, 1991.

[44] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(1):223–255, 1977.

[45] G.D. Plotkin. Domains. Post-graduate lectures in domain theory, Department of Computer Science, University of Edinburgh. Available at hypatia.dcs.qmw.ac.uk/sites/other/domain.notes.other, 1983.

[46] D.B. Plume. A calculator for exact real number computation. BSc Honours Project, University Of Edinburgh, May 1998.

[47] P.J. Potts, A. Edalat, and M.H. Escardó. Semantics of exact real number arithmetic. In *Proceedings of the 12th Annual IEEE Symposium on Logic In Computer Science*, pages 248–257, 1997.

[48] M.B. Pour-el and I. Richards. Computability and non-computability in classical analysis. *Trans. Am. Math. Soc.*, pages 539–560, 1983.

[49] H.G. Rice. Recursive real numbers. *Proc. Amer. Math. Soc.*, pages 784–791, 1954.

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

[51] D.S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript produced in 1969.

[52] A. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 323–342. Springer-Verlag, 1999.

[53] M.B. Smyth. Effectively given domains. *Theoretical Computer Science*, 5(1):256–274, 1977.

[54] M.B. Smyth. Topology. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 641–761. Clarendon Press, Oxford, 1992.

[55] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc., II. Ser.*, 42:230–265, 1936.

[56] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proc. Lond. Math. Soc., II. Ser.*, 43:544–546, 1937.

[57] S. Vickers. *Topology via Logic*. Cambridge University Press, Cambridge, 1989.

[58] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation*, 35(3):233–261, 1993.

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

[60] Klaus W. and C. Kreitz. Type 2 computational complexity of functions on Cantor's space. *Theoret. Comput. Sci.*, 82(1, Algorithms Automat. Complexity Games):1–18, 1991.

[61] K. Weihrauch. Type 2 recursion theory. *Theoretical Computer Science*, 38(1):17–33, 1985.

[62] K. Weihrauch. *Computability*. Springer-Verlag, Berlin, 1987.

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

[64] K. Weirauch. *Computable analysis*. Springer, September 2000.

[65] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.