Topics in the Theory and Practice of Computable Analysis

Branimir Lambov

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Topics in the Theory and Practice of Computable Analysis

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Branimir Lambov
December 16, 2005

Abstract

This dissertation deals with a breadth of computational aspects of analysis, from obtaining computable information from classical proofs in analysis to implementing real number computations efficiently in practice. The main contributions of the paper can be summarized as follows:

- A generalization of a set of conditions given by Matiyasevich under which a recursively defined sequence of real numbers can be shown to converge computably. The generalization replaces monotonicity with a weaker condition and allows errors in the computation of the sequence.
- The first explicit computable bound on the asymptotic regularity of Krasnoselski-Mann iterations with error terms of asymptotically quasi nonexpansive self-mappings of closed convex subsets of uniformly convex Banach spaces. The bound permits the computation of approximate fixed points of the function and relaxes the conditions under which the iteration can be shown to be asymptotically regular.
- An application of the generalization of poly-time computability given by the Basic Feasible Functionals to computable analysis and a correspondence between the resulting notion and the concept of poly-time real-valued function computability as defined by Ko.
- A framework for computable analysis based on interval arithmetic tailored to efficient implementations with methods of including multi-valued real functions in the model and reasoning about the complexity of real functions given by representations in the framework.
- An implementation of exact real arithmetic based on this framework that shows very high efficiency in cases where the precision needed to accurately perform a computation is very low, the set of problems that appear most often in practice and have so far have been the weakness of any exact real number system.
- A package for interval arithmetic with double precision that makes use of the SSE2 set of instructions to provide a very low overhead for interval computations compared to double precision hardware floating point.

Acknowledgements

I would like to express my deepest gratitude to my scientific advisor, Ulrich Kohlenbach, whose incessant stream of information helped me understand the subtleties of many a concept in Mathematical Logic. I want to also thank him for helping me even in pursuing applied results, something not so interesting for a logician.

I want to thank the BRICS scientific and support staff, especially Mogens Nielsen, Uffe Engberg, Karen Kjær Møller, Lene Kjeldsteen, Ingrid Larsen and Lone Moeslund for always being helpful with all the bureaucratic details of being a PhD student in a foreign country, and the lecturers of all the courses I took at BRICS.

Århus was a great place to live and study, and the main reason for this have been the fellow students at BRICS: Bartosz, Dariusz, Frank, Gabriel, Giuseppe, Jesus, Jiri, Kirill, Maciej, Mads, Malgorzata, Paulo, Pawel, Philipp, Saurabh, Sunil.

And finally, I would like to thank the Danish National Research Foundation for giving me this opportunity to study at one of the best PhD schools of the world.

Branimir Lambov,
Århus, December 16, 2005.

Contents

Chapter 1

Introduction

Computable analysis is an area of mathematics as old as computability itself, but an area which does not receive enough attention neither in the practice of computing, where the notion of "real" number is usually taken to be represented by a finite (fixed-size) approximation, nor in mathematical analysis, which claims that finding a monotone bounded sequence is sufficient to provide an algorithm for "computing" a real number.

Both of these views are not sufficient when one is interested in obtaining some forms of useful information about a particular end result.

Fixed-size approximations suffer from the build-up of rounding errors caused by the impossibility to exactly represent operations, meaning that the only information we are certain about is the correctness of the inputs. Nothing is known for certain about the correctness of the result, which often means that one is fooled to accept for accurate results which only have several correct bits, and sometimes made to trust results that have nothing in common with the actual ones (Chapter 8 gives an example of such a result).

The sufficiency of finding a monotone bounded sequence to define a real number in classical mathematics is justified by the least upper bound property of $\mathbb{R}$, which can be used to show that in this case the sequence converges. However, we do not know to what it converges, because it is not possible to obtain a modulus of convergence for the sequence and thereby be able to get approximations of the limit for a given precision. If that information was provided for arbitrary monotone sequences, it could be used to solve the halting problem (see Chapter 2).

As computers are becoming more and more powerful, computable analysis begins to be understood as more than just a theoretical tool. It lets one solve the former problem and gives an indication of the additional information that is required of a mathematical algorithm to provide *computable* results, which provide tools to extract approximations to the number. From the logical side, constructive analysis is a tool that allows one to prove only facts that carry information about a computation. The two are closely related, computable analysis being the language in which the algorithms inherently present in a constructive proof can be formalized and extracted using a suitable proof theoretic technique such as Kleene's functional realizability. Using more elaborate proof theoretic techniques, computable information can also be extracted from

classical proofs that use certain non-constructive principles.

The thesis starts with a chapter that introduces the context, defines the notions of computable real numbers and functions along with some alternative but equivalent formulations, discusses methods to introduce complexity for real numbers and functions, and the topic of intensionality of real-valued functions. It continues with an introduction to proof theory and introduces a method of extracting computable information from non-constructive proofs in analysis.

Chapter 3 of the thesis generalizes a theorem by Matiyasevich showing the computable convergence of certain recursively defined bounded monotone sequences. The monotonicity and boundedness requirements are relaxed to the inequality $0 \leq x_{n+1} \leq (1 + a_n)x_n + b_n$ for bounded in sum sequences (a_n) and (b_n) of non-negative real numbers. This form of generalized monotonicity plays an important role in recent fixed point theory. Additionally, the generalization permits the computation of the sequence to be carried out with errors as long as their sum is bounded asymptotically. The chapter also gives a sample use of the original theorem to a generalization due to Hillam of Kransoselski's fixed point theorem on the real line.

Chapter 4 applies proof mining to a result in metric fixed point theory where Krasnoselski-Mann iterations with error terms of asymptotically quasi nonexpansive mappings of convex subsets of uniformly convex Banach spaces are shown to be asymptotically regular, given that the function has at least one fixed point. We give a bound for the realizer of the Herbrand Normal Form of the statement of asymptotic regularity of the sequence under more general conditions, relaxing the fixed point requirement to an approximate version. The realizer, on one hand, has computational meaning which allows one to compute arbitrarily precise approximate fixed points of the mapping, and, on the other hand, using classical logic it allows one to prove a version of the theorem with the relaxed conditions. To obtain the result, a relaxation of the definition of quasi-nonexpansive functions is introduced, which turns out to be an interesting property on its own right.

In Chapter 2 we introduce complexity of real numbers and functions based on the class of functionals their representation belongs to. In Chapter 5 the complexity class that arises from this definition for the class of the Basic Feasible Functionals is compared to the Oracle Turing Machines running in polynomial time used in Ko's definition of real-valued function complexity. It is shown that for compact intervals the class of basic feasible real functions coincides with the class of the poly-time real-valued functions in the sense of Ko.

Chapters 6 and 7 define a concept of real number and function computability that is constructed to permit very efficient implementations, based on interval computations with a mechanism for increasing precision if needed. Chapter 6 compares it to the definition to real-function computability as given in Chapter 2 and investigates how intensional functions can exist in this model. Chapter 7 shows the obstacles in introducing complexity measures in this framework and defines two kinds of moduli which can be used to show that a real number or function belongs to a certain complexity class.

A very efficient implementation of exact real arithmetic based on this framework, *RealLib*, is discussed in Chapter 8. The library relies on giving its users

two levels of access to real numbers. A top layer which is easier to use and allows one to operate on objects representing complete real numbers, and a bottom layer that operates on approximations but is used as part of the computation of complete real numbers on the top layer. Descriptions of a number of performance problems that appear in implementations are given with the solutions that were used in developing the system. The chapter continues with a description of the system and a performance comparison with other existing packages for real number arithmetic to show that the system is capable of performance in the order of magnitude of the best alternative in all cases, including problems where hardware floating point suffices to produce accurate results.

One of the most important components of this system, a package for interval arithmetic using the SSE2 set of instructions, is discussed in Chapter 9. An explanation of the ideas of the package and the actual implementation of the various basic operations and functions is given. A performance comparison shows that the efficiency of the package surpasses the available alternatives, and the chapter concludes by giving suggestions for simple additional instructions that can be implemented in hardware to further improve the performance at a very low cost.

Appendices A and B form a manual for the *RealLib* package for exact real number computations. Appendix A gives an introduction to the package using a set of examples starting with simple expressions, through transferring the bulk of a computation to the more efficient approximations layer, to a computation of a transcendental function using the mechanisms of the library for computing limits. A detailed reference of the interfaces and functions of the library is given in Appendix B.

Chapter 2

Context

We will assume that the reader is familiar with the concept of computability, its properties and the equivalence of the different formulations of computability, as well as the Church-Turing thesis that states that every machine-implementable procedure can be described by one of the equivalent formulations of computability. The reader should be aware of the known non-computability results, the subrecursive classes (primitive recursive, elementary or poly-time functions) and their definitions ([88] can be used as reference).

Since nothing that has been developed since defies the Church-Turing thesis, we will sometimes use higher-level language concepts (lambda abstraction and application, objects (treatment of countable types, e.g. rational numbers encoded as natural numbers, in the place of positive integers together with sets of operations appropriate for the type) and exceptions (abrupt termination that can pass through an object whose program code we possess)). If the reader is not familiar with the semantics of these concepts, please refer to a modern book on semantics (such as [85]).

For computability of functionals taking function arguments, we will use Hinman's definitions from [39]. The majority of alternative formulations of Type-2 computability are equivalent to this concept, including the Oracle Turing Machines used by Ko [53], the Type-2 Theory of Effectivity used by Weihrauch [113], and Kleene's S1-S9 computability [50] restricted to Type 2.

It is known that strings of natural numbers of fixed length, integers and rational numbers can be encoded in natural numbers in polynomial time in such a way that every natural number denotes a unique element of the encoded set, where also the encodings of $\mathbb{Z}$ and $\mathbb{Q}$ have poly-time basic operations: addition, substraction, multiplication and division, as well as sign extraction and taking the integer part of a rational number (in the sense of truncation, downward rounding, upward rounding or rounding to nearest).

This can be achieved for example by the Cantor pairing $\langle n, m\rangle = (n+m)^2 + m$, encoding integers z by natural numbers n such that

$$z = (-1)^n \left\lceil \frac{n}{2} \right\rceil$$

and rational numbers q by pairs $\langle r, s\rangle$:

$$q = (-1)^r \frac{\lceil r/2 \rceil}{s+1}.$$

2.1 Real numbers and computability

Ordinary computability theory does not easily deal with computations on real numbers because of the infinite nature of real numbers. Since there is a one-to-one correspondence between real numbers and functions on natural numbers, working with real numbers implies being able to generate and process infinite amounts of information.

When the question at hand is to compute a certain real number, our task will be to simply generate an infinite sequence that would identify the number in a given representation. How easy, or whether at all possible, it is to compute a real number depends on the given representation.

One of the most common ways to define the real numbers is via equivalence classes of Cauchy sequences of rational numbers.

2.1.1 Cauchy sequences

Definition 2.1 *A* Cauchy sequence x_n *is a sequence of real numbers such that*

$$\forall \varepsilon > 0 \exists n \in \mathbb{N} \forall m \geq n (|x_n - x_m| < \varepsilon).$$

Unfortunately, for all practical purposes having a computable function able to generate a Cauchy sequence consisting of rational numbers and converging to a given real number, is not sufficient. This is because the definition above does not give us any information about the number in finite time (i.e. we have no way of knowing how far we are from the limit for any given member of the sequence, unless we examine the difference with infinitely many members).

To be able to obtain information, we need to require a witness function for n above:

Definition 2.2 *A* modulus of convergence *for a Cauchy sequence x_n is a function $M : \mathbb{N} \to \mathbb{N}$, such that*

$$\forall k \in \mathbb{N} \forall m > M(k) (|x_m - x_{M(k)}| < 2^{-k}).$$

A modulus of convergence exists for every real number (as it can be obtained by arithmetical comprehension from the definition of a Cauchy sequence), but computable moduli of convergence do not exist even for some numbers that can be given as the limit of a computable Cauchy sequence. A few such numbers are given by the Specker sum [106] $\sum_{i \in A} 2^{-i}$ for a semirecursive A[1].

Using the modulus of convergence to pick a suitable selection of the members of the Cauchy sequence, we obtain the computably equivalent representation via rapidly converging Cauchy sequences:

[1] With a short argument one can show that the computability of such a number implies the recursiveness of A and vice versa. Let A be the range of the recursive function a. Then $x_i = \sum_{j:\exists k \leq i (a(k)=j)} 2^{-j}$ converges to the Specker sum for A.

Definition 2.3 *A* rapidly converging Cauchy sequence *is a sequence of real numbers, such that*

$$\forall k \in \mathbb{N}(|x_k - x_{k+1}| < 2^{-k+1}).$$

The rate 2^{-k+1} is this definition is essential, because it ensures $\forall k \forall m > k(|x_k - x_m| < 2^{-k})$ and thus gives a modulus $M(k) := k$. On the other hand, from a Cauchy sequence x_i with a modulus M one can extract the rapidly converging sequence $x_{M(i+1)}$.

2.1.2 Cauchy function representations

When we talk about computations on real numbers, it is sometimes more convenient to think about approximations of the limit of a sequence rather than forced distances between its elements. Here we give yet another equivalent formulation, which we are going to use as our basic representation of real numbers:

Definition 2.4 *A* Cauchy function representation *(CF-representation) of a real number α is a function $A : \mathbb{N} \to \mathbb{Q}$, such that $\forall n(|A(n) - \alpha| < 2^{-n})$.*

(every rapidly converging Cauchy sequence is a CF-representation of its limit, and shifting the CF-representation by 2 elements gives its limit as a rapidly converging Cauchy sequence)

A computable real number is thus one that has a computable CF-representation. Note that totality is part of the requirement for CF-representations of numbers.

Once we have settled on the representation, functions on real numbers will just have to convert representations of arguments to representations of results. It is straightforward then to come up with this definition of a function on real numbers:

Definition 2.5 *A* Cauchy function representation *(CF-representation) of a function $\phi : \mathbb{R} \to \mathbb{R}$ is a functional $F : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$, such that whenever A is a CF-representation of $\alpha \in$ dom ϕ, $F(A)$ is a CF-representation of $\phi(\alpha)$.*

An important issue must be settled before we can understand what this definition actually means restricted to computable functions, i.e. to settle on what we mean by a 'functional' taking a function as an argument.

This can be settled in two different ways. One of the possibilities is to give the function argument as code of a computable function, avoiding the necessity to go into higher-type recursion theory. This gives rise to a world where the computable real numbers are the only real numbers present (Russian School of Constructive Mathematics). Unfortunately, this invalidates very well known and widely used mathematical principles. In such a world it would not be guaranteed, for example, that a bounded monotone sequence converges, since the limit of a computable sequence of numbers may converge to a non-computable number such as one of the Specker numbers. Even weaker principles, such as Brouwer's fixed point theorem for $\mathbb{R}^2$ or above, would be invalid not only in

the uniform sense, because there exist Lipschitz-continuous and computable mappings on the computable unit square that have no fixed point [91].

Even though both these principles are not constructive, we aim to create a world which makes a framework for computability in classical mathematical analysis and thus should not fail such widely accepted truths. Surely, one could not obtain meaningful information from the use of these principles, but the mere truth of the statements can sometimes be very useful even in extracting computable information (an example of such use of both principles will be given in Chapter 4).

The second possibility is to permit all possible functions as arguments to the CF-representation of a function. This is the approach we are going to take, using some Type-2 notion of computability.

Ordinary recursion theory contains the notion of relative computation, where the behavior of functions operating with a fixed infinite oracle are studied. If we allow the oracle to vary, the concept of relative computation is transformed to computation with function arguments. The only problem is that recursion theory has a requirement for the oracle to be total, and special care needs to be taken to be able to compose real functions and permit partial computations at the same time (which is needed to be able to represent real functions whose domain is not the entire real line).

Problems may arise if we expect functional substitution to recognize whether or not its argument is total. This seems to be a natural requirement for real number computations as a non-total argument does not define a real number. This recognition is not possible and leads to contradictions (see e.g. [39], Exercise 4.27). Instead, we use a restricted form of functional substitution which does not say anything about the behavior of the constructed functional if the argument is not total:

Definition 2.6 ([39], Theorem 3.9) *A functional $F(\underline{n}, \underline{\alpha})$ is constructed via restricted functional substitution from $G(\underline{n}, \underline{\alpha}, \beta)$ and $H(p, \underline{n}, \underline{\alpha})$ if whenever $\lambda p.H(p, \underline{n}, \underline{\alpha})$ is total,*

$$F(\underline{n}, \underline{\alpha}) \simeq G(\underline{n}, \underline{\alpha}, \lambda p.H(p, \underline{n}, \underline{\alpha})).$$

In this thesis the type-2 computability notion we use is Hinman's partial recursive functionals as defined in [39] (following Kleene's definition from [49]), which are closed under this form of functional substitution. Additionally, this definition of restricted functional substitution coincides with the unrestricted version when it is applied to total classes of functionals such as the ones described in Section 2.2.

Composition of two CF-representations F, G of functions ϕ, ψ would require that $F \circ G(A)$ is a CF-representation of $\psi(\phi(\alpha))$ whenever A is a CF-representation of $\alpha \in \text{ dom } \phi$ and $\phi(\alpha) \in \text{ dom } \psi$. This is exactly what the restricted functional substitution gives us.

This is the most we can achieve, as the ideal case where $F \circ G(A)$ is undefined whenever $A \notin \text{ dom } F$, or $F(A) \notin \text{ dom } G$ is not possible (in addition to Hinman's example, the reader may also refer to [113], Exercise 2.1.10).

2.1.3 Computable real numbers and functions

We are now ready to give our definition of computable real numbers and functions:

Definition 2.7 *A real number or function is computable, if it has a CF-representation as a recursive function (resp. partial recursive functional).*

This definition of real functions using the Type-2 notion of functional is able to work on any real number, since all real numbers have CF-representations (although possibly only non-computable ones). The main idea of the representation, to be able to obtain information in finite time, is also preserved.

The latter also implies that to compute finite information about their result, the computable real functions only have access to finite parts of their input. In other words, the computable real functions must all be continuous. In particular, any non-constant function from the real numbers to discrete results cannot be computable. This instantly gives us a few important examples of functions which cannot be fully computable:

- any comparison, especially the equality test;
- any kind of rounding if it requires uniqueness of the result (e.g. first n digits of the infinite decimal representation of a real number).

One can wonder what is the usefulness of the whole idea if we cannot even provide a decimal representation of a result. The solution comes in the form of redundancy: if we permit several choices for the finite size decimal approximations, it is possible to make a function that outputs decimal approximations. More specifically, the so-called 'faithful rounding', where the decimal approximations need to be within a unit in the last place from the real number, can be achieved. Similarly, all graphics where real numbers define objects have to be drawn with "gray areas", pixels for which we do not know if they belong to the object or not.

The last paragraph describes a method of dealing with discontinuity that will be discussed in more detail in Section 2.3. Redundancy is also a key feature of the CF-representations of real numbers. Without redundancy, it is easy to show that simple real function such as addition cannot be implemented (this is the case with Turing's initial idea to use the infinite decimal representation of real numbers to define computability on the reals [111, 112]).

The four arithmetic operations can be easily shown to be computable, defined for example as:

- addition: $+_{\mathbb{R}}(A, B) := \lambda n.A(n+1) +_{\mathbb{Q}} B(n+1)$
- sign change: $-_{\mathbb{R}}(A) := \lambda n. -_{\mathbb{Q}} A(n)$
- multiplication: $\cdot_{\mathbb{R}}(A, B) := \lambda n.A(\text{size}(|B(0)|_{\mathbb{Q}} +_{\mathbb{Q}} 1) + n + 2) \cdot_{\mathbb{Q}} B(\text{size}(|A(0)|_{\mathbb{Q}} +_{\mathbb{Q}} 1) + n + 2)$, where $\text{size}(x) = \mu n < \lceil x \rceil_{\mathbb{Q}}.2^n \geq_{\mathbb{Q}} x$.
- reciprocal:
  $\text{recip}_{\mathbb{R}}(A) := \lambda n.\text{recip}_{\mathbb{Q}}(A(n + 1 + 2\mu k[|A(k)|_{\mathbb{Q}} \geq_{\mathbb{Q}} 2^{-(k+1)}])$

These functionals need to make sure that for a fixed n the result of their application is within 2^{-n} from the actual result of the application of the real number function. To do this, they request higher precision from their arguments. For addition and substraction, this can be done by simply requesting the value of the arguments for $n+1$. Multiplication requires upper bounds for the size of the arguments, which can be obtained by adding 1 to the their value at 0, which then lets us compute an index giving us sufficiently precise approximations. These three functionals are total, while division is not.

To be able to approximate the result of division of one by a real number, we need to find witness that the number is non-zero. This is what the minimization in the functional $\text{recip}_{\mathbb{R}}$ does, looks into the argument until it tells it that the represented number is at least 2^{-k} away from 0. This functional makes many evaluations of the input to find that witness. In particular, if the argument given to $\text{recip}_{\mathbb{R}}$ is 0, the functional will make use of all the information in its argument in order to diverge.

The latter is not a problem as division is not defined for arguments that are real zeroes. For anything that is not zero, $\text{recip}_{\mathbb{R}}$ will eventually find a point where it can be separated from zero, and since the minimization does not depend on n, the reciprocal will return a total function, which is a CF-representation of the reciprocal of the argument.

Unless we have extra information about the argument to division, it is easy to show that unbounded minimization is unavoidable, i.e. unbounded minimization can be defined from a CF-representation of division. If we want to be able to operate on real numbers given as black-box functions, to be able to compute partial real functions we cannot restrict ourselves to a subrecursive class of functionals.

One can write CF-representations for the real numbers and continuous real functions that occur in practice, such as the numbers π, e, Euler's γ, the functions $\sqrt{\cdot}, \log$, $e^{\cdot}$, $\sin$, $\cos, \arctan$, Riemann ζ etc. Some discontinuous functions can be computed if their domains are restricted to exclude the point of discontinuity, e.g. the function

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

The last paragraph gives us reason to believe that the idea is useful not only as a theoretical tool, but also for reliable computations in practice. Additional robustness to the idea is given by the abundance of equivalent alternative formulations.

2.1.4 Alternative formulations

A. Grzegorczyk was the first to define computable analysis in a manner equivalent to the one used today. In [34], he gave several equivalent formulations. In one of them, he represents a real number α as a function $A : \mathbb{N} \to \mathbb{Z}$, such that for all n

$$\left| \alpha - \frac{A(n)}{n+1} \right| < \frac{1}{n+1}.$$

For a function ϕ to be computable on a compact domain with rational endpoints $[a, b]$, he required two things:

- a computable function $F : (\mathbb{Q} \cap [a, b]) \to \mathbb{N} \to \mathbb{Q}$, such that whenever $q \in \mathbb{Q}$, $F(q)$ represents $\phi(q)$.
- a computable modulus of uniform continuity $G : \mathbb{N} \to \mathbb{N}$ for ϕ:

$$\forall n \in \mathbb{N} \forall x, y \in [a, b] \left(|x - y| < \frac{1}{G(n) + 1} \to |\phi(x) - \phi(y)| < \frac{1}{n + 1} \right).$$

This definition avoids any higher-type computability notion in the definition at the expense of having more complicated composition of real functions and application of a computable real function to a real number. The approach is also used by Pour-El and Richards in [93].

A modified version is used by Schwichtenberg in [102], where the numbers are defined as Cauchy sequences of rational approximations with a separate modulus of convergence. The definition of a function is modified accordingly, using three functions, F (with domain as above) generating a Cauchy sequence, H giving a modulus of convergence for the generated sequences uniform in the argument of F, and modulus of uniform continuity G as above. This makes the application operation much simpler as the Cauchy sequence can be constructed using F alone, while the modulus of convergence requires only G and H.

Another one of the formulations given by Grzegorczyk states that a real function ϕ is computable on a compact interval with rational endpoints $[a, b]$ if there exists a computable function $F : I(\mathbb{Q} \cap [a, b]) \to I(\mathbb{Q} \cap [a, b])$ (where by $I(X)$ we denote real intervals with endpoints in X), satisfying

- if $\alpha \in K$ for an interval $K \in I(\mathbb{Q} \cap [a, b])$, then $\phi(\alpha) \in F(K)$
- if $\phi(\alpha) \in L$ for some interval L, then there exists an interval K, such that $F(K) \subseteq L$
- if $K \subseteq L$ are intervals, then $F(K) \subseteq F(L)$.

This formulation allows us to work directly on the level of approximations, does not require any type-2 notion of computability and has very straightforward and simple operations for application and composition. The restriction of a compact domain can be avoided if we permit $\pm\infty$ as possible endpoints of the interval.

The domain theoretic approach to computable analysis used by Edalat [19], Escardo [22] and others uses this formulation, but phrased in domain theoretic terms. The interpretation relies heavily on monotonicity, which is not essential for Grzegorczyk's formulation.

Indeed, Grzegorczyk gives yet another definition using intervals, where a real function ϕ is computable on a compact interval $[a, b]$ if there exists a function $F : I(\mathbb{Q} \cap [a, b]) \to I(\mathbb{Q} \cap [a, b])$ satisfying

- if $\alpha \in K$ for an interval $K \in I(\mathbb{Q} \cap [a, b])$, then $\phi(\alpha) \in F(K)$

- if $\beta \neq \phi(\alpha)$, there exists an interval $K \ni \alpha$, such that $\beta \notin F(K)$.

The second condition states that the output intervals gets arbitrarily small given an appropriately small input interval. This characterization appears to be most useful for efficient practical implementations of exact real arithmetic. It coincides with an approach sometimes used to provide reliable computations, evaluations using interval arithmetic with increasing precision until the output interval shrinks enough to provide a meaningful output. More emphasis on this approach will be given in Chapters 6 and 7.

Ko [53] and Weihrauch [113] use type-2 computability notions and approaches very similar to our definition. Ko uses the concept of Oracle Turing Machines (OTM), which employ a special operation to obtain the value of an oracle (i.e. a type-1 argument) for a given index in one computational step. This is essentially the machine model of the idea of partial recursive functionals that we are using. Ko's model uses the dyadic numbers $\mathbb{D}$ as a base for the reals instead of $\mathbb{Q}$. The dyadic numbers can be defined as the rationals that can be written in the form $m2^n$ where m and n are integers. The dyadic numbers have natural binary representations in the form $\pm 1 a_1 a_2 \dots a_k . b_0 b_1 \dots b_l$, in fact infinitely many since every representation can be extended by adding zeroes to the right. Ko uses $\mathrm{prec}(\pm 1 a_1 a_2 \dots a_k . b_1 b_2 \dots b_l) = l$ as a measure for the length of a binary representation of a dyadic number, and represents a real number α as a function $A : \mathbb{N} \to \mathbb{D}$, such that $\forall n \, (\mathrm{prec}(A(n)) = n \wedge |A(n) - \alpha| \leq 2^{-n})$[2]

A computable real function ϕ in Ko's sense is then defined as an OTM computable $F : (\mathbb{N} \to \mathbb{D}) \to \mathbb{N} \to \mathbb{D}$, such that for every representation A of a real number α, $F(A)$ is a representation of $\phi(\alpha)$.

Weihrauch uses a different but equivalent computability approach (called Type-2 Theory of Effectivity, TTE) where the function arguments are stored on an infinite tape. The machine reads from this tape and writes on a different output tape where it is not allowed to return and delete cells it has already written to. One of the variety of representations that he gives, ρ_C, directly coincides with ours if we translate it to the setting of partial computable functionals. He also gives several equivalent alternatives: ρ, where a real number is described as the list of all intervals containing it and ρ_{sd} where a real number is described via its infinite representation in "signed digit notation", which is a binary representation that uses an additional digit, $\overline{1}$ denoting -1, to introduce sufficient redundancy.

The TTE representation ρ_{sd} and Ko's OTM's working on dyadic Cauchy function representations are especially suitable for complexity analysis.

2.2 Complexity in computable analysis

2.2.1 Complexity classes of real numbers

It is easy to define the complexity of real numbers as the complexity of the function that generates them as a Cauchy function representation. For example,

[2] the difference between $<$ and $\leq$ in the evaluation of the error is inessential as $a < b$ implies $a \leq b$ and $a \leq b$ implies $\frac{1}{2}a < b$ for positive a, b.

the class of the primitive recursive reals can be defined as the real numbers having a primitive recursive CF-representation. This also works for all levels of the Grzegorczyk hierarchy [33] above and including the class of the elementary functions (i.e. the third level of the hierarchy).

For smaller complexity classes such as the poly-time functions, the model of CF-representations above cannot work, because it is impossible to generate a representation that is at least n bits long given an argument n (that is because all poly-time functions of n are bounded by[3] $2^{p(|n|)}$ for some polynomial p, which is not sufficient to generate numbers as big as 2^n). The solution of the problem comes in the form of modified definition of CF-computability, where the rate of convergence of the sequence is $2^{-|n|}$ instead of 2^{-n}.

Definition 2.8 *A* sharp CF-representation *of a real number α is a function $A : \mathbb{N} \to \mathbb{Q}$, such that $\forall n(|A(n) - \alpha| < 2^{-|n|})$.*

This change is immaterial for the higher complexity classes that include the function 2^n, but is essential for the class of the poly-time computable real numbers. We will therefore say that a real number is in a given complexity class if it has a sharp CF-representation in the class.

The situation is more complicated when we consider real functions, because they are higher type objects and there is no universally accepted theory for complexity in Type 2 and above. We will define several classes of subrecursive functionals of higher type that are often used to classify complexity, given in order of decreasing complexity: the levels of Gödel's primitive recursive hierarchy including the primitive recursive functionals in the sense of Kleene S1-S8, the functionals corresponding to the levels of Grzegorczyk's hierarchy, and the Basic Feasible Functionals. To be able to use the latter, we will also use a sharp version of the CF-representations of functions:

Definition 2.9 *A* sharp CF-representation *of a real function ϕ is a functional $F : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$, such that whenever $\alpha \in$ dom ϕ and A is a sharp CF-representation of α, $F(A)$ is a sharp CF-representation of $\phi(\alpha)$.*

2.2.2 Higher-type complexity

The type structure $\mathcal{T}$ is defined inductively by the following two rules:

$$\begin{aligned} 0 &\in \mathcal{T} \\ (\tau \to \sigma) &\in \mathcal{T} \text{ if } \sigma, \tau \in \mathcal{T} \end{aligned}$$

where 0 is the type of the natural numbers $\mathbb{N}$. Other notations for $\tau \to \sigma$ used in the literature include $\sigma(\tau)$ or $(\tau)\sigma$. We will omit brackets if they can be recovered from the right-associativity of the operator $\to$ (i.e. $0 \to 0 \to 0$ means $0 \to (0 \to 0)$, and the brackets in $(0 \to 0) \to 0$ cannot be omitted). We will use either $x : \rho$ or x^ρ to denote that the variable x is of type ρ.

The level of a type is defined by the rule $\text{level}(\tau \to \sigma) = \max(\text{level}(\sigma), \text{level}(\tau) + 1)$ with $\text{level}(0) = 0$. The type of functions on natural numbers $0 \to 0$

[3] as usual in complexity theory, $|n|$ denotes the bit-length of n, i.e. $|n| = \lceil \log_2(n+1) \rceil$.

is of level 1, and the type of functionals $(0 \to 0) \to 0 \to 0$ operating on a function and returning a function (which is equivalent to a functional taking one function and one number argument and returning a number) is of level 2.

The pure types are denoted by integers and defined by induction as 0 and the types $n+1 := n \to 0$ if n is a pure type. The pure type n is a type of level n.

Since lower types are trivially embeddable into higher ones, and arbitrarily long tuples of objects of the same type can be effectively (in polynomial time) coded in a single object, all the types of a certain level can be considered equivalent to the pure type of that level, that is why we will speak directly of type 1 referring to all types of level 1, meaning all functions on an arbitrary number of integer arguments. By type 2 we will thus refer to all functionals that take at least one function argument or generate a type-2 object.

In this thesis the types will be always interpreted over the full set-theoretic type structure S^ω over $\mathbb{N}$:

$$S_0 := \mathbb{N}, S_{\tau\to\rho} := S_\rho^{S_\tau}$$

where $S_\rho^{S_\tau}$ is the set of all set-theoretic functions from S_τ to S_ρ.

Primitive recursive computability in higher types can be given by the terms that can be defined in Gödel's "System $\mathcal{T}$" in the model S^ω. The system was proposed by Gödel in [29] to be used in his functional interpretation of constructive logic. It is defined by the terms constructed from:

- constants 0,
- successor constant S,
- the combinator constant $\Sigma_{\rho,\sigma,\tau}$ for every $\rho, \sigma, \tau \in \mathcal{T}$,
- the projector constant $\Pi_{\sigma,\tau}$ for every $\sigma, \tau \in \mathcal{T}$,
- recursion constant R_ρ for every type ρ.

The meaning of the constants is pretty standard: the constant 0 function, the successor function for natural numbers, the projector and combinator often used as alternative formulation of the typed lambda calculus, and primitive recursion. However, the recursion operation for types other than 0 is not something trivial. The definition of primitive recursion in Gödel's System $\mathcal{T}$ is

$$\begin{aligned} R_\rho xy0 &=_\rho x \\ R_\rho xy(Sz) &=_\rho y(R_\rho xyz)z \end{aligned} \tag{2.1}$$

(where $=_\rho$ is set-theoretic equality, $z : 0, x : \rho$ and $y : \rho \to 0 \to \rho$.)

On type 0 this is just the standard primitive recursion. Recursion on type 1, however, lets one define some non-primitive recursive functions such as the Ackermann function. Moreover, the higher the level of recursion that is used, the more powerful the functions become, which is why we talk about Gödel's primitive recursive hierarchy based on the highest type of the recursion used in their definition.

The functionals of pure type of level 0 in this hierarchy exactly match the functionals that can be defined using Kleene's rules S1 to S8 [50]. At Type 1, they are just the ordinary primitive recursive functions.

One can also define a higher-type extension of Grzegorczyk's hierarchy, which matches the ordinary levels of the hierarchy at Type 1. The higher type Grzegorczyk functionals of level n are defined as the terms built from:

- constants 0,
- successor constant S,
- the combinator constant $\Sigma_{\rho,\sigma,\tau}$ for every $\rho, \sigma, \tau \in \mathcal{T}$,
- the projector constant $\Pi_{\sigma,\tau}$ for every $\sigma, \tau \in \mathcal{T}$,
- constant R_b for bounded recursion

$$\begin{aligned} R_b xyz0 &=_0 x \\ R_b xyz(Sv) &=_0 \min(y(R_b xyzv)v, zv) \end{aligned}$$

  where $x, v : 0$, $y : 0 \to 0 \to 0$, $z : 0 \to 0$

- the n-th branch $A_n : 0 \to 0 \to 0$ of the Ackermann function as defined in [98].

This definition was obtained by removing some redundancies from the definition of $G_n A^\omega$ of [69][4].

A higher-type extension of the polynomial-time computable functions is given by the Basic Feasible Functionals (BFF) [83, 14, 45]. They can be defined as the terms built from:

- a constant for every polynomial-time computable function,
- the combinator constant $\Sigma_{\rho,\sigma,\tau}$ for every $\rho, \sigma, \tau \in \mathcal{T}$,
- the projector constant $\Pi_{\sigma,\tau}$ for every $\sigma, \tau \in \mathcal{T}$,

[4]The additional symbols $\min, \max, \mu_b, \Phi_1, \tilde{R}_\rho$ used in [69] can be recovered using the following definitions:

$$\begin{aligned} P &:= \lambda x.R_b 0(\lambda ab.b)Sx \\ I &:= \lambda xyz.R_b y(\lambda ab.z)(\lambda b.z)x \\ \dot{-} &:= \lambda xy.R_b x(\lambda ab.Pa)(\lambda b.x)y \\ \min &:= \lambda xy.R_b 0(\lambda ab.x)(\lambda b.y)(S0) \\ \max &:= \lambda xy.I(\dot{-}xy)yx \\ \Phi_1 &:= \lambda fx.f(R_b 0(\lambda ab.I(\dot{-}(f(Sb))(fa))a(Sb))Sx) \\ \mu_b &:= \lambda fx.P(R_b 0(\lambda ab.I_m a(I(fxb)(Sb)a)a)S(Sx)) \\ \tilde{R}_\rho &:= \lambda xyzv\underline{w}.R_b(y\underline{w})(\lambda ab.zab\underline{w})(\lambda b.vb\underline{w})x \end{aligned}$$

- constant R_{bn} for bounded recursion on notation:

$$\begin{aligned} R_{bn}xyz0 &=_0 x \\ R_{bn}xyzv &=_0 \min(y(R_{bn}xyz\lfloor v/2\rfloor)v, zv) \end{aligned}$$

  where $x, v : 0$, $y : 0 \to 0 \to 0$, $z : 0 \to 0$

In [45], Kapron and Cook give an Oracle Turing Machine characterization of the Basic Feasible Functionals as the machines that compute in time which is a second-order polynomial in the length of their inputs. A second order polynomial is one that can take function parameters, but a crucial element of this is the definition for a length of a Type-1 argument. The simple definition $|f|(x) = |f(x)|$ is not sufficient, but the definition

$$|f|(x) = \max_{|i|<x} f(i)$$

makes the characterization possible. The latter encompasses the most of the function argument that the BFF functional can make use of.

Just like with real numbers, we will say that a real function belongs to a given complexity class if it has a sharp CF-representation in the class. Since the complexity classes we use are not limited to Type 2, it is very easy to extend the notion to operators on real functions (which would be Type 3 objects).

All of the complexity classes discussed in this subsection can be used to define classes of computable real functions. By construction, they are all closed under composition and unrestricted functional substitution, ensuring that the composition of real functions and the result of the application of a function to a number will remain in the same complexity class. The real numbers and results of functions applied to real numbers will also be part of a well-studied complexity class for type-1 functions[5].

2.2.3 Other characterizations of complexity classes in analysis

Although our concept of complexity for real functions is a straightforward extension of the concept for real numbers, it is not the one commonly used in the computable analysis community, where quantitative evaluations of the number of stages in a machine model of computation are the most accepted method of evaluating complexity.

The main problem that needs to be overcome when one wants to propose a complexity notion for real functions, is the existence of many representations of the same real number of varying quality. In our model of sharp CF-representations it is possible to represent the number one, for example, as:

1. a constant function that returns the same representation of the rational number 1,

2. a function that returns different rational numbers equal to 1, using integer codes of size greater than 2^n,

[5] see [92] for the complexity of primitive recursive functions in Gödel's sense of levels higher than 0.

3. a function that returns $1 - 2^{-(|n|+1)}$, using integer codes of size smaller than $2^{p(|n|)}$ for some polynomial p,

4. a function that produces rational approximations to 1 with the required precision, using integer codes as big as the Ackermann function of $|n|$.

All of these are valid sharp CF-representations of a real number. A polynomial time computable function should be able to process any of them. The ability to do this is an intrinsic feature of all the computability classes we discussed above.

This is not the case if one counts the steps that a Turing machine model for type-2 computability needs in order to compute on the inputs given above. In order to read the outputs of an oracle query for the cases 2 and 4, the Turing machine would require, respectively, non-polynomial and non-primitive recursive time. I.e., should we permit all the possibilities above to represent a real number, there would be

- (due to 2) no bound polynomial in the precision $|n|$ for the running time of any real function that depends on its argument,

- (due to 4) not even a bound primitive recursive in $|n|$.

To solve the problem, one has to reduce the possible speed of growth of the functions representing real numbers. In fact, all real numbers in a bounded interval have dyadic representations, which can be encoded in size smaller than $2^{p(|n|)}$ for the same p. For an unbounded interval, representations of size $2^{p(|n|,|b|)}$ can be given, where b is an upper bound for the represented number. This is the key observation that makes Ko's complexity theory for real functions [53] possible.

The restriction is an integral part of Ko's definition of computable real number. He specifies the dyadic numbers as the base for the reals instead of the rationals, and specifically requires the representations of real numbers to return numbers of precision[6] $|n|$[7], ruling out all but case 3 in the list above.

When the functions are only required to operate on objects of this kind, the running time of an Oracle Turing Machine becomes meaningful:

Definition 2.10 (Ko, [53], 2.18) *Let G be either a bounded closed interval $[a, b]$ or the set $\mathbb{R}$. Let $f : G \to \mathbb{R}$ be a computable function. Then, we say that the* time complexity *of f on $[a, b]$ is bounded by a function $t : G \to \mathbb{N} \to \mathbb{N}$ if there exists an OTM M which computes f in Ko's sense such that for all $x \in G$ and $n > 0$, the running time of $M(x, n)$ is bounded by $t(x, n)$.*

The class of the polynomial-time computable functions $f : [a, b] \to \mathbb{R}$ on a compact interval is then defined as the class of functions whose running time is

[6] Recall from the previous section that the precision of a dyadic number is given by the number of digits to the right of the decimal point.

[7] Although in Section 2.1 of [53], Ko uses n as the argument in the definition of real numbers, in Section 2.2 and throughout the rest of the book, he insists on passing n in unary notation. The latter is equivalent to defining the real numbers as functions of $|n|$.

bounded by a polynomial in the precision n. We will see in Chapter 5 that this definition of polynomial-time computability coincides with BFF-computability of sharp CF-representations on a compact interval.

Alternatively, the poly-time functions in the sense of Ko have a characterization using the first of Grzegorczyk's formalizations of real number computability modified to work with diadic approximations in place of rational numbers:

Definition 2.11 (Ko, [53], 2.21) *A function f is poly-time computable on $[a, b]$ iff there exist poly-time functions m and $\psi : (\mathbb{D} \cap [a, b]) \to \mathbb{N} \to \mathbb{D}$ such that*

- *m is a sharp modulus of continuity for f on $[a, b]$ (i.e. $\forall n \forall x, y \in [a, b](|x - y| \leq 2^{-|m(n)|} \to |f(x) - f(y)| \leq 2^{-|n|})$).*
- *for any $d \in \mathbb{D} \cap [a, b]$ and all $n \in \mathbb{N}$, $|\psi(d, n) - f(d)| \leq 2^{-|n|}$*

It becomes significantly more complicated to define complexity of operators on real functions in Ko's model. The Oracle Turing Machine model only works at type level 2, thus all functions must be translated to a type-1 setting in order to apply it. When real functions are represented in the first of Grzegorczyk's characterizations, functionals of Type 2 can be used to represent real operators. However, the time complexity of these functionals once again becomes useless for the same reason as above, the varying quality of representations of real functions, especially of the modulus of continuity. Ko tries to solve the problem by restricting the class of functions to which the operators can be applied to only polynomial time functions, and verifying whether the application of an operator to a polynomial time function results in a polynomial time function. The resulting notion of pointwise complexity is a meaningful one, but the restrictions are much too severe.

In the TTE model of real number computability, the same quality issue makes the representations ρ and ρ_C useless, but the computably equivalent formulation ρ_{sd} solves the problem by similarly enforcing only representations of size polynomial to the precision. The induced complexity concepts are equivalent to Ko's.

2.2.4 Complexity and interval approaches to computable analysis

In the previous section we gave two of Grzegorczyk's definitions of real number functions that rely on applying functions to intervals with additional requirement that the outputs eventually shrink when they are applied to a sequence of intervals that converges to a single real number, with and without the added requirement of monotonicity.

It is easy to understand why this definition would not easily admit a complexity measure. The word "eventually" is the root of the problem. Since such a function can wait for arbitrarily long time, it is easy to define functions of very low complexity that compute any computable real function (a proof of this will be given in Chapter 7).

A solution of the problem is to replace the word eventually with a specific bound, i.e. to additionally supply a modulus of convergence for real numbers and functions. Just like the case of classical versus computable versions of the notion of Cauchy sequence, here we get a separation between computable and complexity bounded versions. A sequence of shrinking intervals converging to a real number suffices to compute the number, because we can always find the rate of convergence via unbounded minimization. If we want to extract information in complexity bounded time, however, the unbounded minimization is not available, thus we need to have additional information about the number, given in the form of an explicit rate of convergence.

The approach is discussed in more detail in Chapter 7, where we also investigate its relation to sharp CF-computability.

2.3 Intensionality in computable analysis

In the preceding sections we have consciously restricted the objects that represent real functions to output the same real number regardless of the actual representation of the input that was given. Some of the equivalent concepts of Section 2.1 rely on this requirement. It is necessary if we want our objects to represent real number functions.

Sometimes we might like to relax our requirement on the represented objects to something less restrictive than a function. For example, the square root of a complex number can be taken to be one of two equally valid values. To make this into a function, a unique choice between the two values must be made (usually taken as the one with the angle with the positive real line that is smaller in absolute value, and the one with a positive imaginary value if the angles are both $\pi/2$). The choice causes a discontinuity in the function, making it non-computable unless the line of discontinuity is excluded from the domain of the function.

Another example of forcing uniqueness causing discontinuity is the argument function for complex numbers. Here, the possible choices are infinitely many, and the unique selection, as above, picks the least possible absolute value, or π if the number lies on the negative part of real line. The latter becomes a line of discontinuity that must be excluded from the domain if we want to give a computable representation of the function.

Many complex functions may be implemented using the argument function as a building block. This includes the square root example above, as well as raising a complex number to an arbitrary power. For the latter, the straightforward definition $x^y = e^{y \cdot \ln x}$ would be undefined for all x that lie on the negative part of the real line.

Yet another example was already mentioned in Section 2.1: rounding in any form that requires uniqueness. Specifically, the operations $\lfloor \cdot \rfloor$ ('floor function', also known the Gauss staircase), $\lceil \cdot \rceil$ ('ceiling function'), and rounding to the nearest integer are not computable. In Section 2.1 we declared that the problem can be overcome by introducing redundancy in the form of faithful rounding: the faithful rounding of x is allowed to be either $\lfloor x \rfloor$ or $\lceil x \rceil$.

The following function $\mathrm{fr} : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{Z}$ implements faithful rounding in the framework of CF-representations:

$$\mathrm{fr}(A) := \left\lfloor A(1) +_{\mathbb{Q}} \frac{1}{2} \right\rfloor_{\mathbb{Q}}$$

If A represents the real number α, $|A(1) - \alpha| < 2^{-1}$ and thus $\lfloor \alpha \rfloor \leq \mathrm{fr}(A) \leq \lceil \alpha \rceil$.

One can see from this definition that the actual choice that fr makes depends solely on $A(1)$. The same real number may be represented by Cauchy functions with values at 1 that round to a different integer, thus fr does not represent a real number function, because it does not respect equality of real numbers but depends on the actual representation.

In Section 2.1 the rounding was somehow external to the framework, in the sense that we would use the operation in order to finally obtain a meaningful result. As we can see from the discussion above, cases where multiple results are (equally) acceptable arise naturally even for real number functions. It is not hard to see that the functions discussed in the beginning of the section can be implemented in a similar way by a function that depends on the actual representation. It would be thus preferable to permit this mechanism to also work inside the framework for real number computations.

We may say that such CF-functions fail the *extensionality principle* for real numbers, meaning[8] that there exist CF-representations x, y such that $x =_{\mathbb{R}} y$ does not imply $f(x) =_{\mathbb{R}} f(y)$. We will call them *intensional representations* to denote this fact. In the constructive literature some authors use the term *operations* for objects of this kind.

These objects do not describe functions on real numbers. To denote the fact that more than one result is possible for the application of such an object to a real argument, they are usually called "multi-valued functions" in the literature (in [113], for example). To accommodate intensionality, our definition of a real number function must allow for the following:

Definition 2.12 *A CF-representation of a multi-valued real function $\phi : \mathbb{R} \rightrightarrows \mathbb{R}$ is a functional $F : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$, such that whenever A is a CF-representation of $\alpha \in \operatorname{dom} \phi$, $F(A)$ is a CF-representation of a real number $\beta \in \phi(\alpha)$.*

Restricted to (single-valued) functions this coincides with the definition we had before. The discussion above tells us that there are computable multi-valued functions that do not have any computable choice function, i.e. whose computable realizations are all intensional.

Multi-valued functions allow us to avoid discontinuities by replacing points of discontinuity with sufficiently big "buffers" where the function may return results continuous with either the left or right half of the graph. The buffer

[8] If one works in a formal system where real numbers are actually defined by the equivalence classes of CF-representations, this also means that the extensionality principle $\forall f : \mathbb{R} \to \mathbb{R} \forall x, y \in \mathbb{R}(x = y \to f(x) = f(y))$ is indeed false if such objects are permitted

can be made arbitrarily small, as long as it is a neighborhood instead of just a point.

The importance of intensional functions can also be seen from the fact that modifying the Blum/Shub/Smale algebraic model for assessing real function complexity [5] to a version that matches the computable real functions as defined in the thesis relies on some form of approximate comparison. The latter uses the fact that if $a < b$, then for an arbitrary x at least one of the facts $x < b$ or $a < x$ is true and can be shown computably. This can be easily made into a multi-valued function.

The following version is used in Brattka and Hertling's model of feasible real random access machines [6]:

$$<_k (x, y, k) = \begin{cases} \{tt\}, & \text{if } x < y - \frac{1}{k} \\ \{ff\}, & \text{if } x > y + \frac{1}{k} \\ \{ff, tt\}, & \text{otherwise .} \end{cases}$$

Its CF-representation can be given by

$$\mathrm{lt}(X, Y, k) = \begin{cases} tt, & \text{if } X(|k| + 1) \leq_{\mathbb{Q}} Y(|k| + 1) \\ ff, & \text{otherwise .} \end{cases}$$

While intensionality is very easy to introduce in any type-2 framework for analysis (the TTE model, for example, is defined from the beginning for multi-valued functions), it is incompatible with some of the equivalent alternative frameworks that were discussed in Section 2.1. Since the multi-valued functions are only continuous on the representations and not necessarily on the real numbers, the first characterization of Grzegorczyk's does not work, because rational approximations without additional information are not sufficient to *choose* in a way that would make a meaningful definition to replace the uniform modulus of continuity.

The two interval approaches have this information, but would suffer from consistency issues if there is no way to store the choice once it has been made. Chapter 6 will discuss a consistent way to introduce intensionality in an interval approach.

2.4 Classical analysis and computability

How can we use mathematics in this setting of representing real numbers as computable objects? Since we explicitly required a rate of convergence for every Cauchy sequence to make it into a computable real number, and the usual classical tool to extract this information (arithmetical comprehension) is not computable, does ordinary mathematics work if it very often "constructs" a solution to a problem by giving a converging sequence with no known rate of convergence?

In general, the answer to this question is no. However, in many special cases the useful information contained in this kind of "semi-constructive" proof can be recovered. One way to ensure that computable data are given by a proof is the use of constructive logic in place of classical.

In constructive logic every existential quantifier of a true sentence implies a witness (or realizer) for its truth. The proof of an existential statement $\exists x P(x)$, for example, can be used to construct a procedure that computes a witness y which makes $P(y)$ true. In more complex statements the witness is a function, for example the constructive version of the condition "a is a Cauchy sequence" $\forall k \exists n \forall m > n(|a(m) - a(n)| < 2^{-k})$ needs a witness for n. This is exactly the modulus M we talked about in Section 2.1 and required to define real numbers. Therefore, the constructive proof of convergence of a sequence contains in itself everything that is needed to compute the limit as a computable real number.

This correspondence goes much further than just the definition of real numbers (see e.g. [2]), but the information in the constructive proof is not obvious and must be extracted using some proof interpretation. Various proof interpretations will be discussed in the following sections. The process of extracting extra information from existing proofs is called "Proof Mining". In Chapter 4 we will give an application of Proof Mining to fixed point theory.

Via the well known negative translation ([28], see also [110] and [69]), classical logic can be embedded in intuitionistic logic. However, the witnesses contained in this classical fragment have a different meaning from the witnesses in the original proof and their extraction is not an easy task.

Take for example the statement that the sequence of rational numbers a converges:

$$\forall k \exists n \forall m > n(|a(m) -_{\mathbb{Q}} a(n)|_{\mathbb{Q}} <_{\mathbb{Q}} 2^{-k}). \tag{2.2}$$

Its negative translation is the formula

$$\forall k \neg\neg \exists n \forall m > n(|a(m) -_{\mathbb{Q}} a(n)|_{\mathbb{Q}} <_{\mathbb{Q}} 2^{-k}).$$

Some of the proof interpretations which we will speak about in the following chapter cannot extract any information about the realizer of n in this statement. Others can, but, because of the negative results already mentioned, yield something that is not sufficient to constructively recover a realizer of n in the original statement (2.2).

In some cases the original realizer can be recovered using principles that allow us to remove certain double negations, such as the Markov principle M^ω:

$$\forall \underline{x}(\neg\neg \exists \underline{y} A_0(\underline{x}, \underline{y}) \to \exists \underline{y} A_0(\underline{x}, \underline{y})) \tag{2.3}$$

(where by the index $\cdot_0$ we denote a quantifier-free formula, and $\underline{x}, \underline{y}$ are tuples of variables of arbitrary type).

Some interpretations are compatible with this principle, and the exact treatment of the principle makes the difference between having any or no information about the actual witness for classical Π^0_2 statements.

We cannot expect to be able to use similar principles for more complex statements, because there exist classically true Π^0_3 statements that cannot have computable realizers (e.g. a realizer of y in $\forall x \exists y \forall z(Txxy = 0 \vee \neg Txxz = 0$ which is a prenexiation of the classically true fact $\forall x(\phi_x \downarrow \vee \neg \phi_x \downarrow)$ would let us solve the halting problem). Instead, statements of higher complexity must be reformulated, for example to their Herbrand Normal Form, in order to obtain computable output.

Definition 2.13 *The* Herbrand Normal Form *(HNF) of a statement* $A = \forall y_0 \exists x_1 \forall y_1 \ldots \exists x_n \forall y_n A_0(y_0, x_1, y_1, \ldots, x_n, y_n)$ *is defined as*

$$A^H := \forall y_0, f_1, \ldots, f_n \exists x_1, \ldots, x_n A_0(y_0, x_1, f_0(x_1), \ldots, x_n, f_n(x_1, \ldots, x_n)).$$

With the axiom of choice A and A^H are equivalent classically, but not constructively. The HNF works by showing that no counterexample to A exists. Thus it is said that witnesses for $\underline{x}$ above satisfy the no-counterexample interpretation of A. The properties of the HNF of Π^0_3 statements will be further discussed in the next sections.

Herbrand's theorem gives us means to find a finite list of candidate no-counterexample interpretations, but only for sentences given in the so-called "open theories" (where the logical axioms are purely universal). Later Kreisel defines a method [72, 73] of finding recursive functionals that satisfy the no-counterexample interpretation for statements in Peano Arithmetic, but unfortunately one cannot do this easily in a modular way, i.e. to use the no-counterexample interpretations of all the theorems used in a proofs to compute no-counterexample interpretations of the result of that proof requires operations of very high complexity ([60], see also [108]). The ability to achieve this easily is a key feature that a proof interpretation must have in order to be useful in practice.

In the rest of this chapter we will discuss interpretations of the intuitionistic Heyting Arithmetic HA and its higher type versions $\mathrm{E-HA}^\omega$ and $\mathrm{WE-HA}^\omega$ (both are based on Heyting Arithmetic in higher types with equality in higher types as the defined notion

$$\begin{aligned} x =_0 y &:= x = y \\ x =_{\rho\to\tau} y &:= \forall z \in \rho(xz =_\tau yz) \end{aligned}$$

with the former including the full axiom of extensionality

$$\forall z^{\rho_1\to\ldots\to\rho_k\to 0}, x_1^{\rho_1}, y_1^{\rho_1}, \ldots, x_k^{\rho_k}, y_k^{\rho_k} \left(\wedge_{i=1}^k (x_i =_{\rho_i} y_i) \to z\underline{x} =_0 z\underline{y}\right)$$

and the latter is restricted to only the quantifier-free rule of extensionality

$$\frac{A_0 \to s =_\rho t}{A_0 \to r[s] =_\tau r[t]},$$

where $s^\rho, t^\rho, r^{\rho\to\tau}$ are terms of the language and A_0 is quantifier-free. The closed terms definable in these systems correspond to Gödel's primitive recursive functionals.). Proper definitions of the arithmetics and proofs of the properties of the the different interpretations are given in [110] and [69].

2.5 Proof Interpretations

A proof interpretation associates for any given formula in a given formal system another formula in a possibly different system. The interpretation gives a set of transformations that reflect the rules of the system, so that an interpretation

can be given for every formula that can be proved. Out of this interpretation, one can read the realizers (or witnesses) of the result.

The first interpretation we will mention is realizability. When it is applied to $\mathrm{E}-\mathrm{HA}^\omega$, its version is called modified realizability.

Definition 2.14 (Kreisel, [74]) *The* modified realizability interpretation $\underline{x}\ \mathrm{mr}\ A \in L(\mathrm{E}-\mathrm{HA}^\omega)$ *is defined using the following set of rules on the logical structure of the formula* $A \in L(\mathrm{E}-\mathrm{HA}^\omega)$*:*

1. $\underline{x}\ \mathrm{mr}\ A := A$, *if* A *is a prime formula and* $\underline{x}$ *is an empty tuple.*
2. $\underline{x}, \underline{y}\ \mathrm{mr}\ (A \wedge B) := \underline{x}\ \mathrm{mr}\ A \wedge \underline{y}\ \mathrm{mr}\ B$.
3. $z, \underline{x}, \underline{y}\ \mathrm{mr}\ (A \vee B) := \big(((z=0) \to (\underline{x}\ \mathrm{mr}\ A)) \wedge ((z \neq 0) \to (\underline{y}\ \mathrm{mr}\ B)\big)$.
4. $\underline{y}\ \mathrm{mr}\ (A \to B) := \forall \underline{x}(\underline{x}\ \mathrm{mr}\ A \to \underline{y}\underline{x}\ \mathrm{mr}\ B)$.
5. $\underline{x}\ \mathrm{mr}\ (\forall y A(y)) := \forall y(\underline{x}y\ \mathrm{mr}\ A(y))$.
6. $z, \underline{x}\ \mathrm{mr}\ (\exists y A(y)) := \underline{x}\ \mathrm{mr}\ A(z)$.

The interpretation is sound for $\mathrm{E}-\mathrm{HA}^\omega$, i.e. for every closed formula A, if $\mathrm{E}-\mathrm{HA}^\omega \vdash A$ then there exists a tuple of closed terms $\underline{t}$ such that $\mathrm{E}-\mathrm{HA}^\omega \vdash \underline{t}\ \mathrm{mr}\ A$. The soundness theorem (given e.g. in [110], Theorem 3.4.5) gives us rules to construct realizers for every provable formula in $\mathrm{E}-\mathrm{HA}^\omega$. In fact, the soundness theorem is also true for the stronger system $\mathrm{E}-\mathrm{HA}^\omega + AC + IP_\neg + \Gamma$, where AC is the full axiom of choice, $IP_\neg$ is the independence of premise schema for arbitrary negated formulae

$$(\neg A \to \exists y B) \to \exists y(\neg A \to B)$$

and Γ is an arbitrary set of $\exists$-free formulae (an $\exists$-free formula is one built up from prime formulae using only $\wedge$, $\to$ and $\forall$).

If $\underline{t}\ \mathrm{mr}\ A$, one can easily read the specific clause satisfied in a disjunction and the numbers or primitive recursive functions that specify the witnesses of existential quantifiers, but only if they appear non-negated.

As usual in an intuitionistic system, negation is the defined notion $\neg A := A \to (0=1)$. The realizer of a negated formula obtained by modified realizability is empty, since the interpretation only looks for a way to satisfy the conclusion, which as a quantifier-free formula has an empty realizer.

In addition to this, Markov's principle is incompatible with the modified realizability interpretation, since the latter satisfies $IP_\neg$ which yields non-computable results in combination with Markov's principle (indeed Markov's principle contains the instance $\forall x(\neg\neg\exists y Txxy \to \exists y Txxy)$ and applying $IP_\neg$ to it gives $\forall x \exists z(\neg\neg\exists y Txxy \to Txxz)$, a realizer of which would solve the halting problem). Therefore we cannot expect to be able to use modified realizability on classical results easily.

Modified realizability can be used to extract information from constructive proofs. There are even machine-implemented modified realizability systems

that can analyze completely formalized proofs and extract realizers automatically. Additionally, the Friedman A-translation [23, 15] combined with modified realizability is able to extract information from classical proofs, but a more general approach should deal with the problem at its roots by defining a better proof interpretation.

The problem in dealing with negated statements comes from the fact that modified realizability only looks into the positive part of implications, but does not say anything about how a failure in the conclusion leads to a failure in the premise. Since the definition of negation in intuitionistic logic relies on this mechanism, negation is not properly treated by realizability.

Gödel's functional interpretation treats implication in a manner that exposes this mechanism.

Definition 2.15 (Gödel, [29]) *The* functional interpretation *of a formula* $A \in L(\mathrm{WE}-\mathrm{HA}^\omega)$ *is the translation* $A^D \equiv \exists \underline{x} \forall \underline{y} A_D(\underline{x}, \underline{y})$ *in the same language. The free variables of* A^D *are those of* A*, the types and length of* $\underline{x}$ *and* $\underline{y}$ *depend on the logical structure of* A *only, and* A_D *is a quantifier-free formula.*

$$
\begin{array}{rrcl}
(i) & A^D & := & A_D := A \text{ for a prime formula } A, \\
(ii) & (A \wedge B)^D & := & \exists \underline{x}\underline{u} \forall \underline{y}\underline{v} (A \wedge B)_D \\
 & & := & \exists \underline{x}\underline{u} \forall \underline{y}\underline{v} (A_D(\underline{x}, \underline{y}) \wedge B_D(\underline{u}, \underline{v})), \\
(iii) & (A \vee B)^D & := & \exists z, \underline{x}\underline{u} \forall \underline{y}\underline{v} (A \vee B)_D \\
 & & := & \exists z, \underline{x}\underline{u} \forall \underline{y}\underline{v} ((z = 0 \to A_D(\underline{x}, \underline{y})) \wedge (z \neq 0 \to B_D(\underline{u}, \underline{v}))), \\
(iv) & (\exists z A(z))^D & := & \exists z, \underline{x} \forall \underline{y} (\exists z A(z))_D \\
 & & := & \exists z, \underline{x} \forall \underline{y} A_D(\underline{x}, \underline{y}, z), \\
(v) & (\forall z A(z))^D & := & \exists \underline{X} \forall z, \underline{y} (\forall z A(z))_D \\
 & & := & \exists \underline{X} \forall z, \underline{y} A_D(\underline{X} z, \underline{y}, z), \\
(vi) & (A \to B)^D & := & \exists \underline{U}, \underline{Y} \forall \underline{x}, \underline{v} (A \to B)_D \\
 & & := & \exists \underline{U}, \underline{Y} \forall \underline{x}, \underline{v} (A_D(\underline{x}, \underline{Y x v}) \to B_D(\underline{U x}, \underline{v})).
\end{array}
$$

The translations of $\neg A$ and $\neg\neg A$ that this interpretation induces do expose information:

$$
\begin{aligned}
(\neg A)^D &= \exists \underline{Y} \forall \underline{x} \neg A_D(\underline{x}, \underline{Y x}) \\
(\neg\neg A)^D &= \exists \underline{X} \forall \underline{Y} \neg\neg A_D(\underline{XY}, \underline{Y}(\underline{XY}))
\end{aligned}
$$

In other words, the functional interpretation of the negation of a statement is a functional that takes candidate proofs of A_D and finds counterexamples to them. The functional interpretation of a classical statement refutes the fact that a candidate mechanism for finding counterexamples works by providing a witness to the contrary.

The latter is very similar to Kreisel's no-counterexample interpretation. In a specific instance, let us examine the functional interpretation of the negative translation of the statement (2.2) that the sequence of rational numbers a converges:

$$
\begin{aligned}
&(\forall k \neg\neg \exists n \forall m \geq n (|a_m - a_n| < 2^{-k}))^D \\
&= \forall k \exists N \forall M (|a_{N(M)} - a_{N(M)+M(N(M))}| < 2^{-k})
\end{aligned}
$$

$$= \exists N \forall k \forall M(|a_{N(M,k)} - a_{N(M,k)+M(N(M,k))}| < 2^{-k}). \tag{2.4}$$

This is exactly the HNF of the statement (2.2), but unlike the no-counterexample interpretation, the functional interpretation is modular in the sense that from the functional interpretation of A and $A \to B$ one can always easily construct the functional interpretation of B, which is highly non-trivial for the no-counterexample interpretation when the implication has a more complex logical structure. Already for Σ^0_3 the functional interpretation of the negative translation differs from the no-counterexample interpretation, and the higher-type objects used in the definition of the functional interpretation make it possible to have simple interpretation of the modus ponens rule.

The statement (2.4) is a constructively weaker form of convergence that can be satisfied computably for the Specker sequences. In fact, since the sequences that converge to these numbers are monotone and bounded, using the following lemma one can obtain primitive recursive realizers for (2.4) for any Specker number:

Lemma 2.1 *Let (a_n) be a sequence in $\mathbb{R}_+$ with $a_{n+1} \leq a_n$ for all n. Then*[9]

$$\forall \varepsilon > 0 \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \max_{i < \lfloor a_0/\varepsilon \rfloor} g^i(0) \left(a_n - a_{g(n)} \leq \varepsilon\right).$$

While this form does not allow us to compute limits of sequences that have been proven classically to converge as computable real numbers (which is impossible because of the Specker examples), it does let us extract *some* information about the limits. In certain cases one is interested only in the distance between a_n and a_{n+1} (for example in applying Krasnoselski-Mann iterations to a function to find an approximate fixed-point), and this is given by the HNF for $M(n) := n+1$ (see Chapter 4). In some cases additional information combined with the HNF allows us to recover the full constructive statement of convergence ([61]).

The soundness theorem for functional interpretation ([110], Theorem 3.5.10 (ii)) is valid for the system $\mathrm{WE} - \mathrm{HA}^\omega + AC + M^\omega + IP_\forall + \Pi$, where M is Markov's principle for all finite types, $IP_\forall$ is the independence of premise principle for universal statements

$$(\forall \underline{x} A_0(\underline{x}) \to \exists y B(y)) \to \exists y (\forall \underline{x} A_0(\underline{x}) \to B(y))$$

(where y is of arbitrary finite type and does not occur free in the quantifier-free A_0), and Π is a collection of purely universal statements. It lets one extract closed terms $\underline{t}$, such that $\mathrm{WE} - \mathrm{HA}^\omega + \Pi \vdash \forall \underline{y} A_D(\underline{t}, \underline{y})$ whenever $\mathrm{WE} - \mathrm{HA}^\omega + AC + M^\omega + IP_\forall + \Pi \vdash A$, where A is a sentence of $\overline{L}(\mathrm{WE} - \mathrm{HA}^\omega)$.

Both the restriction to weak extensionality and of the independence of premise principle to only universal statements are essential, since the combination of Markov's principle in all higher types with full extensionality or more general IP principle leads to contradictions with the possibility to extract computable realizers (see [43] and also [69]).

[9] The notation g^i stands for g iterated i times.

If we extend $\mathrm{WE}-\mathrm{HA}^\omega$ to the classical system $\mathrm{WE}-\mathrm{PA}^\omega$ by adding the law of excluded middle $A \vee \neg A$ for arbitrary formulae, we can state the following version of the soundness theorem: whenever $\mathrm{WE}-\mathrm{PA}^\omega + \mathrm{QF}-\mathrm{AC} + \Pi \vdash A$ (where QF-AC is the quantifier-free axiom of choice, $\forall \underline{x} \exists \underline{y} A_0(\underline{x}, \underline{y}) \to \exists \underline{Y} \forall \underline{x} A_0(\underline{x}, \underline{Y}\underline{x})$), one can extract closed terms $\underline{t}$, such that $\mathrm{WE}-\mathrm{HA}^\omega + \Pi \vdash \forall \underline{y}(A')_D(\underline{t}, \underline{y})$, where A' is a negative translation of the sentence A.

For Π^0_2 statements the existence of a computable realizer is always guaranteed, because it can be found via unbounded minimization. The functional interpretation gives us more than just that, since it interprets Markov's principle by looking in the computational information that is present in the functional interpretation of a doubly negated statement. Unlike some forms of realizability which interpret Markov's principle by unbounded search, functional interpretation does not require unbounded minimization to satisfy Markov's principle, and can therefore be used for complexity-bounded fragments of arithmetic ([14, 59]) where all representable functionals match one of the complexity classes defined in Section 2.2.

The instance of the soundness of the functional interpretation for $\mathrm{WE}-\mathrm{PA}^\omega$ for Π^0_2-statements can be given by the following theorem:

Theorem 2.1 ([69]) *If* $\mathrm{WE}-\mathrm{PA}^\omega + \mathrm{QF}-\mathrm{AC} + \Pi \vdash \forall \underline{x} \exists \underline{y} A_0(\underline{x}, \underline{y})$*, where* $\underline{x}$ *and* $\underline{y}$ *are the only free variables in the quantifier-free* A_0*, then there exists a tuple of closed terms* $\underline{t}$ *of* $\mathrm{WE}-\mathrm{HA}^\omega$*, such that* $\mathrm{WE}-\mathrm{HA}^\omega + \Pi \vdash \forall \underline{x} A_0(\underline{x}, \underline{t}\underline{x})$.

An even more interesting modification is obtained when one combines the functional interpretation with a mechanism that allows one to reason in bounds instead of actual realizers. A requirement for this is some notion of monotonicity in higher types which allows one to preserve bounds across applications of functionals and therefore permits one to achieve modularity of bounds via the modus ponens rule.

2.6 Majorizability and its combination with proof interpretations

Definition 2.16 (W.A. Howard, [41]) *We define* $x^* \ \mathrm{maj}_\rho \ x$ *for a finite type* ρ *by induction on the type:*

$$x^* \ \mathrm{maj}_0 \ x := x^* \geq x,$$

$$x^* \ \mathrm{maj}_{\tau\to\rho} \ x := \forall y^*, y \left(y^* \ \mathrm{maj}_\tau \ y \to x^*(y^*) \ \mathrm{maj}_\rho \ x(y) \right).$$

We will say that a class of functionals C *is hereditarily majorized by another class* C^* *if for every functional* $F \in C$ *there exists* $F^* \in C^*$ *such that* $F^* \ \mathrm{maj} \ F$*. We will also use the term "hereditarily self-majorized" for the classes that are majorized by themselves.*

Majorizability is closely linked with complexity. In fact, all the complexity classes discussed in Section 2.2 are hereditarily self-majorized. Here we will give a short proof of this fact for the class of the Basic Feasible Functionals

following the proofs in [41] and [69] of the self-majorizability of the levels of Gödel's primitive recursive hierarchy and the Grzegorczyk higher-type hierarchy, respectively.

Theorem 2.2 *For every functional $F \in \boldsymbol{BFF}$ there exists $F^* \in \boldsymbol{BFF}$, such that F^* maj F.*

Proof. $\Sigma_{\delta,\rho,\tau}$ and $\Pi_{\rho,\tau}$ are self majorizable, and for every poly-time function f there exists a polynomial p with coefficients among the natural numbers, such that $f(\underline{x}) \leq 2^{p(|\underline{x}|)} = f^*(\underline{x})$. The right hand side of this inequality is a polytime function for which $\forall \underline{x} \forall \underline{y} \leq \underline{x} \left(f^*(\underline{x}) \geq f^*(\underline{y}) \geq f(\underline{y})\right)$, i.e. f^* maj_1 f. Define

$$R^*_{bn}(x, y, g, h) := h(x)$$

If $x^*, y^* \geq x, y$, g^* $\mathrm{maj}_{0\to 0\to 0}$ g and h^* maj_1 h

$$R^*_{bn}(x^*, y^*, g^*, h^*) = h^*(x^*) \geq h(x) \geq R_{bn}(x, y, g, h),$$

which proves R^*_{bn} $\mathrm{maj}_{0\to 0\to (0\to 0\to 0)\to 1\to 0}$ R_{bn}.

Now the theorem follows from the fact that t^* $\mathrm{maj}_{\rho\to\tau}$ $t \wedge s^*$ maj_ρ s implies $t^* s^*$ maj_τ ts. □

The monotone functional interpretation is obtained by combining the two notions, replacing the construction of exact realizers in the soundness theorem by the construction of their majorizers, which in some cases (such as the rule $A \to A \wedge A$) is a much simpler task. The soundness theorem of the monotone functional interpretation gives rules to construct terms $\underline{t}^*$, such that

$$\exists \underline{x}(\underline{t}^* \text{ maj } \underline{x} \wedge \forall \underline{y} A_D(\underline{x}, \underline{y})).$$

In an existential statement the realizer is a constant, thus having a majorizer for it simply gives us a bound. In a statement of the form $\forall x \in \mathbb{N} \exists y \in \mathbb{N}$ the realizer of y is a function $Y : \mathbb{N} \to \mathbb{N}$ and its majorizer is a monotone function which is everywhere greater than Y, i.e. a complexity bound for Y.

The soundness theorem for monotone functional interpretation ([58]) is valid for $\mathrm{WE} - \mathrm{HA}^\omega + AC + M^\omega + IP_\forall + \Delta$, where Δ is a set of sentences that have the logical form $\forall \underline{a} \exists \underline{b} \leq \underline{r}\underline{a} \forall c B_0(\underline{a}, \underline{b}, \underline{c})$ for some tuple of closed terms $\underline{r}$, giving us rules to extract majorizers $\underline{t}^*$ such that $\mathrm{WE} - \mathrm{HA}^\omega + AC + \Delta \vdash \exists \underline{x}(\underline{t}^* \text{ maj } x \wedge \forall \underline{y} \forall \underline{a} A_D(\underline{x}\underline{a}, \underline{y}, \underline{a}))$ whenever $\mathrm{WE} - \mathrm{HA}^\omega + AC + M^\omega + IP_\forall + \Delta \vdash A(\underline{a})$.

The main benefit of using the combination is that ineffective principles that belong to the set Δ have realizers in monotone functional interpretation which allow us to use these principles in extracting computational information from proofs that use them, namely bounds on the final results. One of these principles is the Weak König's Lemma (WKL) stating that every infinite binary tree has an infinite branch. We will not detail the method which translates WKL to the appropriate form here, the reader can refer to [55] for details.

As we did in the previous section, we can state a version of the soundness theorem for the classical system $\mathrm{WE} - \mathrm{PA}^\omega$ applied to Π^0_2 sentences:

Theorem 2.3 ([58]) *Let τ be a type of level at most 2, ρ be an arbitrary finite type, and s be a closed term of* $\mathrm{WE}-\mathrm{PA}^\omega$. *If* $\mathrm{WE}-\mathrm{PA}^\omega+\mathrm{QF}-\mathrm{AC}+\Delta\vdash \forall x^1\forall y\leq_\rho sx\exists \underline{z}^\tau A_0(x,y,z)$, *where the only free variables in A_0 are x, y and z, then there exists a closed term $\underline{t}$, such that* $\mathrm{WE}-\mathrm{HA}^\omega+AC+\Delta\vdash \forall x\forall y\leq_\rho sx\exists z^\tau\leq_\tau txA_0(x,y,z)$.

In the following section we will describe a concrete result achieved using this mechanism.

2.7 Proof Mining and Analysis

Analysis is a very good candidate for Proof Mining, because a lot of information is hidden in the use of real numbers as basic objects. When one specifies the representations of real numbers as in Section 2.1, information that is not obvious in a proof is exposed. Simple statements in analysis start to involve quantifiers, for example the equality test $x=y$ for $x,y\in\mathbb{R}$ becomes $\forall n(|X(n)-_{\mathbb{Q}}Y(n)|_{\mathbb{Q}}<_{\mathbb{Q}}2^{-(n-1)}$, where X and Y are the representations of the two numbers. Quantification over the real numbers becomes quantification over the type-1 functions with an universal test.

If, for example, we apply this to a statement of the form

$$\forall x,y\in\mathbb{R}(f(x)=0\wedge f(y)=0\to x=y) \tag{2.5}$$

we transform the statement to

$$\forall X^1,Y^1(\mathbb{R}(X)\wedge\mathbb{R}(Y)\to(F(X)=_{\mathbb{R}}0\wedge F(Y)=_{\mathbb{R}}0\to X=_{\mathbb{R}}Y))$$

(where F is a CF-representation of the function f and $\mathbb{R}(X)$ is true if X is a CF-representation of a real number) and after we apply the definitions of $=_{\mathbb{R}}$ and $\mathbb{R}(x)$

$$\forall X^1,Y^1((\forall p\forall q>p(|X(p)-_{\mathbb{Q}}X(q)|_{\mathbb{Q}}<_{\mathbb{Q}}2^{-p}\wedge|Y(p)-_{\mathbb{Q}}Y(q)|_{\mathbb{Q}}<_{\mathbb{Q}}2^{-p}))\to \tag{2.6}$$
$$(\forall n(|F(X)|_{\mathbb{R}}(n)<_{\mathbb{Q}}2^{-n}\wedge|F(Y)|_{\mathbb{R}}(n)<_{\mathbb{Q}}2^{-n}\to\forall k(|X-_{\mathbb{R}}Y|)_{\mathbb{R}}(k)<_{\mathbb{Q}}2^{-k}))).$$

This statement is too complicated to gather any useful information from its proof interpretation. However, one can avoid the external implication by ensuring $\mathbb{R}(X)$ and $\mathbb{R}(Y)$, or rather by finding a conversion which we will denote by $\tilde{X}$ that makes sure that $\tilde{X}$ is a real number for every X^1. By also ensuring that $\tilde{X}=_1X$ whenever X is indeed a real number, we can make sure that role of the quantifiers $\forall X\in\mathbb{R}$ and $\exists Y\in\mathbb{R}$ will be fulfilled by $\forall X^1$, resp. $\exists Y^1$ and substituting $\tilde{X}$ and $\tilde{Y}$ for X and Y.

The following computable[10] functional implements the $\tilde{\cdot}$ operator:

$$\tilde{f}(k)=\begin{cases} f(k),\ \text{if } \forall n,n'<k(|f(n)-_{\mathbb{Q}}f(n')|_{\mathbb{Q}}<_{\mathbb{Q}}2^{-n}) \\ f(\mu m<k.[\exists n,n'<m(|f(n)-_{\mathbb{Q}}f(n')|_{\mathbb{Q}}\geq_{\mathbb{Q}}2^{-n})]),\ \text{otherwise}\end{cases} \tag{2.7}$$

[10]In fact, it is easy to see that the modification of this functional to sharp CF-computability belongs to all complexity classes discussed in Section 2.2.

When we apply the substitution to (2.5), we obtain the much more suitable for proof mining form

$$\forall X^1, Y^1 \forall k \exists n(|F(\tilde{X})|(n) <_{\mathbb{Q}} 2^{-n} \wedge |F(\tilde{Y})|(n) <_{\mathbb{Q}} 2^{-n} \to |\tilde{X} - \tilde{Y}|(k) <_{\mathbb{Q}} 2^{-k}). \tag{2.8}$$

In fact, we may assume that the $\tilde{\cdot}$ operator is included in every operation on real numbers, i.e. we can define e.g. $X -_{\tilde{\mathbb{R}}} Y := \tilde{X} -_{\mathbb{R}} \tilde{Y}$ and $\tilde{F}(X) := F(\tilde{X})$ and just assume $\tilde{\mathbb{R}}$ (resp. $\tilde{F}$ etc.) is used everywhere instead of $\mathbb{R}$ (resp. F).

The statement (2.8) is a Π^0_2 statement for which we know that we can find a realizer for n as a function of X, Y and k. However, the form above talks about the representations of the numbers, while in practice we would prefer to ignore the actual representations and talk about how approximate a root a number has to be in order to know that it is an approximation of the root. In other words, we want to have the following version of the above statement:

$$\forall X^1, Y^1 \forall k \exists n(|F(X)| <_{\mathbb{R}} 2^{-n} \wedge |F(Y)| <_{\mathbb{R}} 2^{-n} \to |X -_{\mathbb{R}} Y| <_{\mathbb{R}} 2^{-k}).$$

The latter would again complicate the structure too much, but we can observe that in the statement above any of the $<_{\mathbb{R}}$ relations may be equivalently substituted for their non-strict versions[11]. We may then change the comparisons, so that the prenexiation of both of them results in an existential quantifier, i.e.

$$\forall X^1, Y^1 \forall k \exists n(|F(X)| \leq_{\mathbb{R}} 2^{-n} \wedge |F(Y)| \leq_{\mathbb{R}} 2^{-n} \to |X -_{\mathbb{R}} Y| <_{\mathbb{R}} 2^{-k}). \tag{2.9}$$

Exposing the hidden quantifiers in the latter statement gives us

$$\forall X^1, Y^1 \forall k \exists n \exists p \exists q$$

$$(|F(X)|(p) <_{\mathbb{Q}} 2^{-n} +_{\mathbb{Q}} 2^{-p} \wedge |F(Y)|(p) <_{\mathbb{Q}} 2^{-n} +_{\mathbb{Q}} 2^{-p}$$
$$\to |X -_{\mathbb{R}} Y|(q) <_{\mathbb{Q}} 2^{-k} +_{\mathbb{Q}} 2^{-q}),$$

where we can choose to ignore the realizers of p and q (which would be quite messy and tell us facts we do not need to know). As long as the comparisons are of the form that matches the last quantifier in the statement, we can ignore the representation details and analyze the proof up to the form (2.9), which in fact gives us exactly what we want to know: the realizer of n in (2.9) is known as a modulus of uniqueness for the root of the function F and can be obtained from a proof of uniqueness of that root (see [56]).

If x is to come from a bounded interval $[-b, b]$ of $\mathbb{R}$ we can apply the technique we used to avoid the test if X represents a real number to define $\hat{X}$ to convert all X^1 to representations of numbers in the bounded set, also ensuring that the encoding of $\hat{X}$ is majorized by some function b^* of b (see [56] for details on the actual construction of $\hat{\cdot}$ and b^*, or Chapter 5 for a very similar construction).

[11] since $x < 2^{-n} \to x \leq 2^{-n} \to x < 2^{-(n-1)}$, and $\forall n(x < 2^{-(n-1)})$ is equivalent to $\forall n(x < 2^{-n})$

Using monotone functional interpretation and the fact that b^* maj $\hat{X}$ for all X^1 we will be able, for example, to obtain bounds for the above realizers that do not depend on x at all, but only on the bound. Moreover, it would be possible to analyze proofs that use non-effective principles such as the Weak König's Lemma by substituting computable bounds for the possibly non-computable result of the application of the principle.

The latter appears in analysis in the following forms [104]:

- every uniformly continuous function attains its maximum on a compact interval,
- every continuous function on a compact interval is uniformly continuous,
- every continuous function can be uniformly approximated by polynomials of rational coefficients on a compact interval,
- the Heine-Borel theorem on $[0, 1]$ (every covering by a sequence of open intervals has a finite subcovering),
- Brouwer's fixed point theorem (every continuous mapping of $[0, 1]^n$ to itself has a fixed point),
- Schauder's fixed point theorem for separable Banach spaces,
- Hahn-Banach theorem for separable Banach spaces,
- Cauchy-Peano existence theorem.

2.7.1 Abstract spaces

When one tries to extract computational content from proofs in analysis that deal with a general class of spaces (such as e.g. metric spaces satisfying some additional properties such as hyperbolic spaces, CAT(0) spaces etc.), sometimes it is beneficial to abstract away the details about the space and its elements. To do this, one can introduce a system that is based on a hierarchy of types built upon two basic types, $\mathbb{N}$ and X where X is the space in question. For the objects of the class X we define a metric $d : X \to X \to 1$, which gives a representation of a real number for the distance between two objects in X. We can then define equality of objects in X as the defined notion $x =_X y := d(x, y) =_{\mathbb{R}} 0$ and equality in higher types as before.

Using the separation of X from the numeric types one can make a very general framework for analysis, where the only computational information given by objects of type X is the distance between such objects or (in a normed space) the norm of the objects. In extracting bounds from such a framework in a bounded space, one can completely replace any appearance of objects of the space by a bound on the size of the space.

Indeed, U. Kohlenbach proves a series of metatheorems that guarantee certain numerical results can be achieved from a proof of a fact in certain logical systems. One of the instances of these metatheorems applies to the setting of bounded convex subsets of uniformly convex[12] normed spaces, where the logical

[12]See Definition 2.18 below for a definition of the concept of uniform convexity.

framework is given by the system $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$, where $\mathcal{A}^\omega$ is a system for analysis based on the extension of the classical system $\mathrm{WE-PA}^\omega$ to the type hierarchy over $\mathbb{N}$ and X, where equality in X is a defined notion. The system includes the axiom of dependant choice, a function $\|\cdot\| : X \to 1$ that gives the norm of objects in X as a representation of a real number, a characteristic function χ_C for the bounded convex subset C and a modulus of uniform convexity η. An axiom stating that the subset C is bounded by $b \in \mathbb{Q}$ is explicitly included in the framework. The system $\mathcal{A}^\omega$ is powerful enough to formulate most proofs in analysis.

Definition 2.17 *A formula F is called $\forall$-formula (resp. $\exists$-formula) if it has the form $F \equiv \forall \underline{a}^\sigma F_{qf}(\underline{a})$ (resp. $F \equiv \exists \underline{a}^\sigma F_0(\underline{a})$) where $\underline{a}^\sigma = a_1^{\sigma_1}, \ldots, a_k^{\sigma_k}$, F_0 does not contain any quantifier and the types in σ_i are $\mathbb{N}$ or C.*

Theorem 2.4 ([67]) *Let η be a constant of type 1, and $\tau = C, \mathbb{N} \to C$ or $C \to C$. s is a closed term of type $1 \to 1$ and $B_\forall, C_\exists$ are $\forall$- resp. $\exists$-formulas. If the sentence*

$$\forall x^1 \forall y \leq_1 s(x) \forall z^\tau \big(\forall u^0 B_\forall(x, y, z, u) \to \exists v^0 C_\exists(x, y, z, v)\big)$$

is provable in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$, then one can extract a computable functional $\Phi : 1 \to 0 \to 1 \to 0$ of type level 2 in S^ω such that

$$\forall y \leq_1 s(x) \forall z^\tau \big[\forall u \leq \Phi(x, b, \eta)\, B_\forall(x, y, z, u) \to \exists v \leq \Phi(x, b, \eta)\, C_\exists(x, y, z, v)\big]$$

holds in any non-trivial (real) uniformly convex normed linear space $(X, \|\cdot\|)$ with convexity modulus η and any non-empty b-bounded convex subset $C \subset X$.
Instead of single variables x, y, z, u, v we may also have finite tuples of variables $\underline{x}, \underline{y}, \underline{z}, \underline{u}, \underline{v}$ as long as the elements of the respective tuples satisfy the same type restrictions as x, y, z, u, v.
Moreover, instead of a single premise of the form '$\forall u^0 B_\forall(x, y, z, u)$' we may have a finite conjunction of such premises.

This theorem is applicable to recent theorems in fixed-point theory dealing with the convergence of Krasnoselski-Mann iterations of asymptotically quasi-nonexpansive mappings. Before we analyze how it can be applied, we will give a very short introduction to the subject.

2.7.2 Nonexpansive self-mappings and Krasnoselski iterations

In fixed point theory (for an introduction, see [46]), two basic theorems stand on opposite sides of the spectrum, Schauder's and Banach's fixed point theorems.

Theorem 2.5 (Schauder's fixed point theorem) *Let K be a nonempty compact convex subset of a Banach space X, and suppose $f : K \to K$ is continuous. Then f has at least one fixed point.*

(This theorem is a generalization of Brouwer's fixed point theorem to arbitrary Banach spaces.)

Theorem 2.6 (Banach's contraction mapping principle) *Let (X, d) be a non-empty complete metric space. Let $f : X \to X$ be a contraction (i.e. Lipschitzian function with constant $q < 1$). Then f has a unique fixed point $\tilde{x}$ in X and for each $x \in X$, $\lim_{n\to\infty} f^n(x) = \tilde{x}$. Moreover, $d(f^n(x), \tilde{x}) \leq \frac{q^n}{1-q} d(f(x), x)$.*

The latter applies to a very restricted class of functions, but gives computable iterative procedure and a computable bound for the construction of the fixed point, while the former applies to a very large class of functions, but gives no indication of procedure for computing a fixed point. Furthermore, as we already mentioned in Section 2.1, even for $\mathbb{R}^2$ and the square $[0, 1] \times [0, 1]$ there exist computable real functions without a computable fixed point ([91]). In [103] it is shown that Schauder's fixed point theorem is equivalent in complexity to the Weak König's Lemma.

When we try to consider an extension of the class of functions for which we can construct procedures to find a fixed point to the class of nonexpansive mappings (with Lipschitz constant 1) we run into three major problems:

- the space has to be bounded, otherwise mappings such as $f : \mathbb{R} \to \mathbb{R}, f(x) := x + 1$ do not have fixed points;
- if a fixed point exists, it may be not unique, e.g. $f : [0, 1] \to [0, 1], f(x) := x$.
- even in the case of unique fixed points, the Banach iterative procedure will possibly not lead to it. For example, consider $f : [0, 1] \to [0, 1], f(x) := 1 - x$ on the unit interval of the reals: it has the unique fixed point $1/2$, but the iterative procedure started at 0 will keep oscillating between 0 and 1.

Some algebraic structure for the space is needed to obtain results in this case, thus we replace arbitrary metric spaces by normed ones. The first results in that part of the field use a condition on normed spaces called *uniform convexity*:

Definition 2.18 ([11]) *A normed linear space $(X, \|\cdot\|)$ is uniformly convex if*

$$\forall \varepsilon > 0 \exists \delta > 0 \forall x, y \in X (\|x\|, \|y\| \leq 1 \wedge \|x - y\| \geq \varepsilon \to \|\frac{1}{2}(x + y)\| \leq 1 - \delta)$$

A function $\eta : (0, 2] \to (0, 1]$ providing such δ for a given ε is called a modulus of uniform convexity.

An example of uniformly convex space is the space $\mathbb{R}^2$ with the ordinary Euclidean norm (with $\eta(\varepsilon) = \varepsilon^2/8$); a space that is not uniformly convex is again $\mathbb{R}^2$, but with the norm $\|(x_1, x_2)\| = \max(|x_1|, |x_2|)$, because one can pick two different points in the unit ball (which in this case looks geometrically as a square), whose convex combination does not lie in the interior of the ball.

With this definition in place the following theorem gives us a procedure to find a fixed point:

Theorem 2.7 (Krasnoselski, [71]) *Let K be a non-empty convex compact set in a uniformly convex Banach space $(X, \|\cdot\|)$ and f a nonexpansive self-mapping of K. Then for every $x_0 \in K$, the sequence*

$$x_{k+1} = \frac{x_k + f(x_k)}{2}$$

converges to a fixed point $z \in K$ of f.

For example, the very first Krasnoselski iteration started at 0 finds the fixed point of the function $f(x) = 1 - x$ above.

This type of result has been focus of much attention and generalizations of Krasnoselski's theorem have been given in a number of ways, which will be discussed below. Additionally, Ishikawa [42] gives a proof of the statement without a requirement for uniform convexity of the space, but for the generalized classes of mappings all results rely on this property (see [27]). Browder and Petryshyn, on the other hand, realize that the compactness of K is not required if we restrict the result to the so-called "asymptotic regularity" of the sequence x_n, i.e. the fact that $\|x_n - f(x_n)\| \to 0$ (one of the uses of compactness in the theorem is the finding of a fixed point through Schauder's theorem which can be replaced by an application of a result by Browder, Göhde and Kirk [8, 30, 47], and the other use of compactness is required to imply the convergence of the sequence (x_n) given its asymptotic regularity, which cannot be avoided):

Theorem 2.8 ([9]) *Let K be a non-empty convex closed and bounded set in a uniformly convex Banach space $(X, \|\cdot\|)$ and f a nonexpansive self-mapping of K. Let $\lambda \in (0, 1)$. Then for every $x_0 \in K$, the iteration*

$$x_{i+1} = (1 - \lambda)x_i + \lambda f(x_i)$$

satisfies

$$\|x_n - f(x_n)\| \to 0.$$

Although this result does not give a fixed point of the function f, it still gives one approximate fixed points, which can be argued to be sufficient in practice. However, the actual *computable* information given by Kransoselski's result is nothing more than what is given by the statement above, because it does not give us any computable procedure to construct the fixed point (see [68] for a counterexample). Since Krasnoselski's theorem applies to a very restricted class of spaces, results closer to 2.8 are clearly better to analyze for additional computable information.

More general classes of functions for which we know that some form of the Krasnoselski iterations is asymptotically regular are given by the following classes.

Definition 2.19 ([26]) *$f : C \to C$ is said to be asymptotically nonexpansive with sequence $(k_n) \in [0, \infty)^{\mathbb{N}}$ if $\lim_{n\to\infty} k_n = 0$ and*

$$\|f^n(x) - f^n(y)\| \leq (1 + k_n)\|x - y\|, \ \forall n \in \mathbb{N}, \forall x, y \in C.$$

Example 2.1 (taken from [26]) *: Let B is the unit ball in the Hilbert space l^2 and let f be defined as*

$$f((x_1, x_2, x_3, \ldots)) = (0, x_1^2, A_2 x_2, A_3 x_3, \ldots)$$

where A_i is a sequence of numbers $0 < A_i < 1$ such that $\prod_{i=2}^{\infty} A_i = 1/2$. Then F is Lipschitzian with $\|f(x) - f(y)\| \leq 2\|x - y\|$, $x, y \in B$; moreover, $\|f^i(x) - f^i(y)\| \leq 2\prod_{j=2}^{i} A_j \|x - y\|$ for $i = 2, 3, \ldots$. Thus

$$\lim_{i\to\infty} k_i = \lim_{i\to\infty} 2\prod_{j=2}^{\infty} A_j - 1 = 0.$$

Thus f is asymptotically nonexpansive, but clearly not nonexpansive.

Definition 2.20 ([18]) *$f : C \to C$ is quasi-nonexpansive if*

$$\|f(x) - p\| \leq \|x - p\|, \ \forall x \in C, \forall p \in Fix(f).$$

Example 2.2 (taken from [18]) *$f : \mathbb{R} \to \mathbb{R}$,*

$$f(x) = \begin{cases} 0, & \text{if } x = 0 \\ (x/2)\sin(1/x), & \text{otherwise .} \end{cases}$$

The only fixed point is $f(0) = 0$, and $\|f(x) - f(0)\| = \|f(x)\| \leq \|x/2\| \leq \|x - 0\|$. On the other hand, $\|f(\frac{2}{3\pi}) - f(\frac{1}{2\pi})\| = \frac{1}{3\pi} > \frac{1}{6\pi} = \|\frac{2}{3\pi} - \frac{1}{2\pi}\|$. Hence f is quasi-nonexpansive, but not nonexpansive.

A logical combination that covers both definitions above is the notion of asymptotically quasi-nonexpansive mappings:

Definition 2.21 *$f : C \to C$ is asymptotically quasi-nonexpansive with sequence $k_n \in [0, \infty)^{\mathbb{N}}$ if $\lim_{n\to\infty} k_n = 0$ and*

$$\|f^n(x) - p\| \leq (1 + k_n)\|x - p\|, \ \forall n \in \mathbb{N}, \forall x \in X, \forall p \in Fix(f). \tag{2.10}$$

The literature uses a more generalized scheme for iterations known as Krasnoselski-Mann iterations:

Definition 2.22 (Krasnoselski-Mann iterations)

$$x_0 := x \in C, \ x_{n+1} := (1 - \lambda_n)x_n + \lambda_n f(x_n), \lambda_i \in [0, 1]$$

The form used for asymptotically nonexpansive mappings replaces $f(x_n)$ by $f^n(x_n)$ (see e.g. [101]). Additionally, one can account for errors in the computation by including an error term and define the Krasnoselski-Mann iterations with error terms (introduced in [114]):

$$x_0 := x \in C, \ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n,$$

where $\alpha_n, \beta_n, \gamma_n \in [0, 1]$ with $\alpha_n + \beta_n + \gamma_n = 1$ and $u_n \in C$ for all $n \in \mathbb{N}$.

Finally, we define an additional property which is often needed to treat asymptotically non-expansive mappings:

Definition 2.23 ([101]) *$f : C \to C$ is said to be uniformly λ-Lipschitzian ($\lambda > 0$) if*

$$\|f^n(x) - f^n(y)\| \leq \lambda\|x - y\|, \ \forall n \in \mathbb{N}, \forall x, y \in C.$$

2.7.3 Application

A result that combines all the generalizations from above can be given by the following theorem (which combines the results of a series of recent papers, see [94, 114, 101] and most closely [96]):

Theorem 2.9 *Let $(X, \|\cdot\|)$ be a uniformly convex (real) normed linear space and C be a bounded convex subset of X. Let $k \in \mathbb{N}$ and $\alpha_n, \beta_n, \gamma_n \in [0,1]$ such that $1/k \leq \beta_n \leq 1 - 1/k$, $\alpha_n + \beta_n + \gamma_n = 1$ and $\sum \gamma_n < \infty$. Let $f : C \to C$ be a uniformly Lipschitz continuous function which is asymptotically quasi-nonexpansive and has at least one fixed point. Define*

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n,$$

where (u_n) is a bounded sequence in C. Then the following holds:

$$\|x_n - f(x_n)\| \to 0.$$

Assuming the compactness of C the condition of having a fixed point is directly satisfied by Schauder's theorem. In a more general context where the space is Banach and C is closed in addition to being convex and bounded, but the mappings are restricted to asymptotically non-expansive, a fixed point can be shown to exist by Goebel and Kirk's fixed point theorem [26], which relies on the uniform convexity of the space. The given form encompasses both cases, and also allows one to weaken the requirements as we will see below. Additionally, not aiming for a fixed point of the function (which as we already mentioned is not possible to find computably) but settling for the asymptotic regularity of (x_n) allows us to treat X as an abstract space and use the ideas of the section to restrict the appearance of objects in X in the extracted bounds only as bounds on the size of the space.

To apply any quantitative reasoning, we must make explicit the various hidden bits of information in the statement. The uniform Lipschitz continuity requires a number l such that $\forall n \forall x, y \in C(\|f^n(x) - f^n(y)\| \leq l\|x - y\|)$. The asymptotic quasi non-expansiveness requires a sequence (k_n) which satisfies (2.10) together with a bound K for $\sum_{n=0}^{\infty} k_i$. Additionally, we ask for an explicit bound b on the size of the convex subset C, u on the norm of the error sequence (u_n), and E on $\sum_{n=0}^{\infty} \gamma_n$.

The statement of Theorem 2.9 can be given as the following logical statement:

$$\forall f : C \to C \forall k \in \mathbb{N} \forall (k_n) : \mathbb{N} \to [0,1] \forall (\alpha_n), (\beta_n), (\gamma_n) : \mathbb{N} \to [0,1] \forall (u_n) : \mathbb{N} \to C$$

$$\forall l \in \mathbb{Q}_+ \forall b \in \mathbb{Q}_+ \forall u \in \mathbb{Q}_+ \forall E \in \mathbb{Q}_+ (A \to B)$$

where A is the conjunction of the following conditions:

$$A_1 : \quad \forall n (1/k \leq_{\mathbb{R}} \beta_n \leq_{\mathbb{R}} 1 - 1/k)$$

$$A_2 : \quad \forall n (\sum_{i=0}^{n} k_n \leq_{\mathbb{R}} K)$$

$$A_3: \quad \forall n(\sum_{i=0}^{n} \gamma_n \leq_{\mathbb{R}} E)$$

$$A_4: \quad \forall n(\|u_n\| \leq_{\mathbb{R}} u)$$

$$A_5: \quad \forall n \forall x, y \in C(\|f^n(x) - f^n(y)\| \leq_{\mathbb{R}} l\|x - y\|)$$

$$A_6: \quad \forall n(\alpha_n + \beta_n + \gamma_n =_{\mathbb{R}} 1)$$

$$A_7: \quad \exists p(\|f(p) - p\| =_{\mathbb{R}} 0)$$

$$A_8: \quad \forall n \forall p, x \in C(\|f(p) - p\| =_{\mathbb{R}} 0 \to \|f^n(x) - f^n(p)\| \leq_{\mathbb{R}} (1 + k_n)\|x - p\|)$$

and B is the statement

$$\forall q \exists n \forall m > n(\|x_m - f(x_m)\| \leq_{\mathbb{R}} 2^{-q}). \tag{2.11}$$

In fact A_6 is not necessary, because we can entirely omit it by using substitute sequences $\tilde{\beta}_n = \beta_n, \tilde{\gamma}_n = \max(\gamma_n, 1 - \tilde{\beta}_n), \tilde{\alpha}_n = 1 - \tilde{\beta}_n - \tilde{\gamma}_n$ in order to force the sum to be always 1 without affecting the truth of the other conditions involving (γ_n) and (β_n). There are hidden quantifiers in the comparisons above, but they are all $\forall$-statements, thus they do not add up to the logical complexity of the formula with the exception of the premise in A_8. In fact that statement is incompatible with Theorem 2.4 and we cannot change it using one of the tricks discussed in previous sections to put it in the proper logical form.

Instead, we can observe that the proof of Theorem 2.9 (implicitly given in the analysis in Chapter 4) does not use quasi-nonexpansiveness in its full. We can prove a slightly modified version of the theorem which only requests the function to be asymptotically non-expansive with respect to only a certain fixed point of the function. This lets us replace A_8 with a weaker form of quasi-nonexpansiveness that applies only to some fixed point of the function:

Definition 2.24 *A function $f : C \to C$ is called* asymptotically weakly quasi-nonexpansive *if it satisfies*

$$\exists p \in C \forall n \forall x \in C(\|f(p) - p\| =_{\mathbb{R}} 0 \wedge \|f^n(x) - f^n(p)\| \leq_{\mathbb{R}} (1 + k_n)\|x - p\|).$$

This form also absorbs A_7 and allows us to pull $\exists p \in C$ to the list of universal quantifiers in front of the implication, leaving a $\forall$-statement for the premise. Because quasi-nonexpansiveness is used only in cases where the existence of a fixed point is additionally present, this notion is weaker. It is also more general (e.g. the function $x^2 : [0, 1] \to [0, 1]$ is only quasi-nonexpansive in the weak sense), but appears to be sufficient to carry out many results. Shortly after the appearance of [66], the same notion appears in an equivalent form as "J-type" mappings in [24] where the relevant fixed point of the function is called a "center" (see also [25]).

With this change the theorem we analyze assumes the following form:

Theorem 2.10 *Let $(X, \|\cdot\|)$ be a uniformly convex (real) normed linear space and C be a bounded convex subset of X. Let $k \in \mathbb{N}$ and $\alpha_n, \beta_n, \gamma_n \in [0, 1]$ such that $1/k \leq \beta_n \leq 1 - 1/k$, $\alpha_n + \beta_n + \gamma_n = 1$ and $\sum \gamma_n < \infty$. Let $f : C \to C$ be*

a uniformly Lipschitz continuous function which is asymptotically weakly quasi non-expansive. Define

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n,$$

where (u_n) *is a bounded sequence in* C*. Then the following holds:*

$$\|x_n - f(x_n)\| \to 0.$$

It remains to handle the conclusion. It is a Π^0_3 statement of convergence, which we cannot directly use in an application of Theorem 2.4. However, once we observe that changing the non-strict comparison in it to a strict version does not change the meaning of the statement, and replace it with its Herbrand Normal Form

$$\forall q \forall M^1 \exists n(\|x_{n+M(n)} - f(x_{n+M(n)})\| <_{\mathbb{R}} 2^{-q}),$$

we obtain a statement which does have the necessary structure to allow a universal quantifier to be pulled out of the implication to leave the $\exists$-statement as the conclusion.

After these changes the theorem

$$\forall f : C \to C \forall p \in C \forall \underline{s} \in [0,1]^{\mathbb{N}} \forall (u_n) \in C^{\mathbb{N}} \forall \underline{b} \in \underline{\mathbb{Q}} \forall q^0 \forall M^1$$

$$(A_1 \wedge \ldots \wedge A_5 \wedge \|f(p) - p\| =_{\mathbb{R}} 0 \wedge \forall n \forall x \in C(\|f^n(x) - f^n(p)\| \leq_{\mathbb{R}} (1 + k_n)\|x - p\|)$$
$$\to \exists n(\|x_{n+M(n)} - f(x_{n+M(n)})\| <_{\mathbb{R}} 2^{-q}))$$

(where we have shorthanded the list of $[0,1]$ sequences by $\underline{s}$ and the list of bounds by $\underline{b}$) has the necessary logical form to apply Theorem 2.4. We are guaranteed to achieve bounds on both the existential statement in the conclusion, a quantitative result which allows us to find a bound on the time it takes to reach an approximate fixed point of f, and on a selection of the universal statements in the premise, which can be used to obtain a qualitative strengthening of the result by relaxing the condition on p being a fixed point by the availability of arbitrarily precise approximate fixed points.

The metatheorem 2.4 guarantees that we can find a functional Φ primitive recursive in the sense of Gödel such that

$$\forall f : C \to C \forall p \in C \forall \underline{s} \in [0,1]^{\mathbb{N}} \forall (u_n) \in C^{\mathbb{N}} \forall \underline{b} \in \mathbb{Q} \forall q^0 \forall M^1$$

$$(A_1 \wedge \ldots \wedge A_5 \wedge \forall n \leq \Phi(q, M, \underline{b}) | \|f(p) - p\|(n)| <_{\mathbb{Q}} 2^{-n} \wedge$$
$$\forall n \forall x \in C(\|f^n(x) - f^n(p)\| \leq_{\mathbb{R}} (1 + k_n)\|x - p\|)$$
$$\to \exists n \leq \Phi(q, M, \underline{b})(\|x_{n+M(n)} - f(x_{n+M(n)})\| <_{\mathbb{R}} 2^{-q}))$$

holds. The bound Φ does not depend on the bounded parameters $\underline{s}$, but only on some of their features given as bounds in $\underline{b}$. Moreover, the bound is completely uniform in f as long as it satisfies the requirements, and in p, which in particular lets us formulate a stronger version of the original theorem[13]:

[13] using the classical step of recovering $\|x_n - f(x_n)\| \to 0$ from its HNF

Theorem 2.11 *Let $(X, \|\cdot\|)$ be a uniformly convex (real) normed linear space and C be a bounded convex subset of X. Let $k \in \mathbb{N}$ and $\alpha_n, \beta_n, \gamma_n \in [0,1]$ such that $1/k \leq \beta_n \leq 1 - 1/k$, $\alpha_n + \beta_n + \gamma_n = 1$ and $\sum \gamma_n < \infty$. Let $f : C \to C$ be a uniformly Lipschitz continuous function which has arbitrarily precise approximate fixed points p_ε for all $\varepsilon > 0$, such that*

$$\forall n \forall x \in C(\|f^n(x) - f^n(p_\varepsilon)\| \leq_{\mathbb{R}} (1 + k_n)\|x - p_\varepsilon\|).$$

Define

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n,$$

where (u_n) is a bounded sequence in C. Then the following holds:

$$\|x_n - f(x_n)\| \to 0.$$

In case the mapping is asymptotically nonexpansive, this version no longer requires Goebel and Kirk's fixed-point theorem. Instead, one can use much simpler reasoning to satisfy the conditions of this theorem (Lemma 4.6) which is also true for incomplete normed spaces and does not require the subset C to be closed.

The actual extraction of the bound can be done automatically if we have a completely formalized proof of the theorem in the system in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$, but in practice it is easier to perform the bound extraction by hand using the proof of the metatheorem as guidance, rather than trying to formalize the theorem. The extraction is performed in Chapter 4, where we achieve all that the metatheorem predicts, and even the following stronger result, which does not require boundedness of the whole subset C:

Theorem 2.12 *Let X, x, (x_n), $(\alpha_n), (\beta_n), (\gamma_n), (u_n)$ be as above. Let C be a convex subset of X and $f : C \to C$ be uniformly l-Lipschitzian and*

$$\forall \varepsilon > 0 \exists p_\varepsilon \in C \left(\begin{array}{l} \|f(p_\varepsilon) - p_\varepsilon\| \leq \varepsilon \ \wedge \\ \|p_\varepsilon - x\| \leq d \ \wedge \\ \forall y \in C \forall n \left(\|f^n(y) - f^n(p_\varepsilon)\| \leq (1 + k_n)\|y - p_\varepsilon\|\right) \end{array} \right) \quad (2.12)$$

where $d \in \mathbb{Q}^*_+, k_n \in \mathbb{R}_+$ and also $\sum_{n=0}^{\infty} k_n \leq K \in \mathbb{Q}_+$.

Let $1/k \leq \beta_n \leq 1 - 1/k$ for some $k \in \mathbb{N}$, $\sum_{n=0}^{\infty} \gamma_n \leq E \in \mathbb{Q}_+$, and (u_n) be bounded with $\|u_n - x\| \leq u \in \mathbb{Q}_+$.

Then

$$\forall \delta \in (0,1] \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \Phi \forall m \in [n, n + g(n)] \left(\|x_m - f(x_m)\| \leq \delta\right)$$

where $\Phi = \Phi(K, E, u, k, d, l, \eta, \delta, g)$ and

$$\begin{array}{rcl} \Phi(K, E, u, k, d, l, \eta, \delta, g) & = & h^i(0) \\ h & = & \lambda n.(g(n+1) + n + 1) \\ i & = & \left\lfloor \frac{3(5KD + 6E(U+D) + D)k^2}{\varepsilon\eta(\varepsilon/(D(1+K)))} \right\rfloor \\ D & = & e^K(d + EU) \\ U & = & u + d \\ \varepsilon & = & \delta/(2(1 + l(l+1)(l+2))). \end{array}$$

While in other cases of proof extraction from similar theorems one can obtain a realizer for the full statement of convergence (for example, using monotonicity of $\|x_n - f(x_n)\|$, see [61]), this does not appear to be the case here. Still the HNF of the statement is sufficient to obtain useful information, such as a bound on the number of iterations we need to perform in order to obtain an approximate fixed point of the mapping f by using $\lambda n.0$ for the parameter g (Corollary 4.6).

The weak form of asymptotical quasi-nonexpansiveness is essential to obtaining the form (2.12). Trying to weaken the stronger notion to approximate fixed points leads to much stronger conditions which, unlike (2.12), are not satisfied by the conditions of Theorem 2.11 or 2.9. Naturally, the metatheorem does not allow this.

2.7.4 Refined Metatheorems

The actual extraction of the bound provides a more general result than what Metatheorem 2.4 predicts. Subsequently, by defining majorizability in the space X parametrized by an element of the space, Gerhardy and Kohlenbach [25] give refined versions of the set of metatheorems that replace the requirements for boundedness of the convex set (by dropping the axiom stating that C is bounded, which is denoted by the index $_{-b}$ in the designation of the formal system below) with a local version that only accounts for the distances between the elements used in the definition of the theorem. The definition of generalized strong majorizability that allows this result is the following:

Definition 2.25 ([25], Definition 3.3) *We define the relation $\gtrsim^a_\rho$ between objects x, y, a of type $\hat{\rho}$, ρ and X respectively (where $\hat{\rho}$ is obtained from ρ by replacing all instances of X with 0) by induction on ρ as follows:*

$$\begin{aligned} x^0 \gtrsim^a_0 y^0 &:= x \geq_{\mathbb{N}} y, \\ x^X \gtrsim^a_X y^X &:= x \geq_{\mathbb{R}} d_X(y, a), \\ x \gtrsim^a_{\rho\to\tau} y &:= \forall z', z(z' \gtrsim^a_\rho z \to xz' \gtrsim^a_\tau yz) \wedge \forall z', z(z' \gtrsim^a_{\hat{\rho}} z \to xz' \gtrsim^a_{\hat{\tau}} xz). \end{aligned}$$

In case the subject theorem deals with normed spaces, the 0 used implicitly by the norm operation must also be accounted for, thus the above operation is used relative to the point $a := 0$. The following theorem can now correctly predict the conditions from which a computable realizer of the HNF of the statement of asymptotic regularity can be given:

Theorem 2.13 (instance of Theorem 6.3 of [25]) *Let η be a constant of type 1, and $\tau = C, \mathbb{N} \to C$ or $C \to C$. s is a closed term of type $1 \to 1$ and $B_\forall, C_\exists$ are $\forall$- resp. $\exists$-formulas.*
If the sentence

$$\forall x^1 \forall y \leq_1 s(x) \forall z^\tau \big(\forall u^0 B_\forall(x, y, z, u) \to \exists v^0 C_\exists(x, y, z, v)\big)$$

is provable in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]_{-b}$, then one can extract a computable functional[14] *$\Phi : 1 \to 1 \to 1 \to 0$ of type level 2 such that*

$$\forall y \leq_1 s(x) \forall z^\tau \forall z^* \gtrsim^0_\tau z$$

[14] The second argument to Φ is of type $\hat{\tau}$, which in this case is at most type 1.

$$\big[\forall u \leq \Phi(x, z^*, \eta)\, B_\forall(x, y, z, u) \to \exists v \leq \Phi(x, z^*, \eta)\, C_\exists(x, y, z, v)\big]$$

holds in any non-trivial (real) uniformly convex normed linear space $(X, \|\cdot\|)$ *with convexity modulus* η *and any non-empty convex subset* $C \subseteq X$*.*

Instead of single variables x, y, z, u, v *we may also have finite tuples of variables* $\underline{x}, \underline{y}, \underline{z}, \underline{u}, \underline{v}$ *as long as the elements of the respective tuples satisfy the same type restrictions as* x, y, z, u, v*.*

Moreover, instead of a single premise of the form '$\forall u^{\mathbb{N}} B_\forall(x, y, z, u)$*' we may have a finite conjunction of such premises.*

Only one detail in the actual realizers is not directly ensured by this refined metatheorem, the fact that the realizers depend only on bounds on the distances between the objects involved instead of bounds on their norm. Since the normed space X allows one to talk about the norm of separate elements, if one is to treat completely generic proofs dealing with normed spaces, one cannot avoid requesting a bound for the norm of the points in the space involved in the definition of the theorem.

In fact, Theorem 2.11 requires X to be a normed space only to formulate uniform convexity. The fact that the latter and the norm operation are only used on differences between two elements in the space is the reason why our manual bound extraction does not require a bound for the size of one of the elements used in the definition (in this case the most obvious choice is a bound on x). It is very probable that the theorem can be stated for a hyperbolic space which satisfies Machado's axioms that characterize convex subsets of normed spaces [79], in which case a different metatheorem ([25], 4.10) would apply and ensure that the realizers depend on only the distances between the objects used in the definition of Theorem 2.11.

Chapter 3

Rates of convergence of recursively defined sequences

This chapter is a reprint of [76], *"Rates of convergence of recursively defined sequences"*, Proceedings of the 6th Workshop on Computability and Complexity in Analysis (CCA 2004), Electronic Notes in Theoretical Computer Science, Volume 120 (2005), pp. 125-133.

Abstract

This paper gives a generalization of a result by Matiyasevich which gives explicit rates of convergence for monotone recursively defined sequences. The generalization is motivated by recent developments in fixed point theory and the search for applications of proof mining to the field. It relaxes the requirement for monotonicity to the form $x_{n+1} \leq (1+a_n)x_n+b_n$ where the parameter sequences have to be bounded in sum, and also provides means to treat computational errors.

The paper also gives an example result, an application of proof mining to fixed point theory, that can be achieved by the means discussed in the paper.

3.1 Introduction

In classical mathematics many interesting results are based on the fact that

every bounded monotone sequence of real numbers converges to a finite limit. (3.1)

Unfortunately, this fact is not reflected constructively, i.e. there is no theorem letting us compute the speed of the convergence of the sequence which would be needed to compute the limit. Moreover, as shown by Specker in [106], it is possible to construct computable monotone sequences whose limit is non-computable, thus finding the speed is even impossible in certain cases.

If we are interested in extracting effective data from a proof that uses this fact, this imposes a significant problem. Since we do not have a constructive analog for it, we would generally be unable to continue past an application of it. The quantitative information hidden within the proof may thus seem inaccessible.

Sometimes it is possible to bypass this problem using (constructively) weakened versions of the statement of convergence like its Herbrand Normal Form (HNF):

$$\forall k \in \mathbb{N} \forall g : \mathbb{N} \to \mathbb{N} \exists n (g(n) \geq n \to |x_n - x_{g(n)}| \leq 2^{-k}). \tag{3.2}$$

This modification is not strong enough to allow the Specker examples, and we even have a solution for the general theorem in the form of a functional (for $u \leq x_i \leq x_{i+1} \leq v$ for all i)

$$H(u, v, k, g, x) \equiv \mu i \leq (v - u)2^k \left(g^i(0) < g^{i-1}(0) \vee |x_{g^{i-1}(0)} - x_{g^i(0)}| \leq 2^{-k} \right). \tag{3.3}$$

By certain proof-theoretic techniques (a combination of negative translation and an appropriate so-called monotone version of Gödel's functional interpretation, see [58]) one can show that in extracting bounds from proofs based on (3.1) it is sufficient to maintain bounds for (3.2) throughout the proof if the conclusion has a sufficiently simple logical form or is weakened accordingly. This is due to the fact that having a bound on (3.2) is nothing else than having a realizer for the monotone functional interpretation of the negative translation of (3.1).

The weakened form may be sufficient to get a full rate of convergence. E.g. [61] treats a proof in fixed point theory that uses the fact (3.1), but does not require its full power as even an instance of its HNF with $g(n) = n + 1$ suffices to yield the final result. In other cases (e.g. [66]) the weakened form appears in the final result, but may be sufficient to extract valuable information.

However, this approach does not always give the needed answer. It is thus interesting to investigate whether other approaches can resolve the ineffectivity caused by the use of this fact without weakening it. Naturally, these would rely on additional information about the converging sequence.

An approach to handling this problem was taken by Matiyasevich in [81]. He addressed the question of the convergence of a bounded monotone sequence which is defined recursively. It is known that if a diagonalization of the function that defines the recursion has a unique root in the interval where the sequence lies, then the sequence converges to that root. Matiyasevich proved the following theorem:

Theorem 3.1 *Let (x_n) be a non-decreasing sequence of real numbers from the segment $[u, v]$ and let F be a uniformly continuous function defined on all r real numbers $\langle y_1, y_2, \ldots, y_r \rangle$ such that $u \leq y_1 \leq y_2 \leq \ldots \leq y_r \leq v$ with modulus of continuity ω, i.e.*

$$\forall k \forall x_0 \leq x_1 \leq \ldots \leq x_{r-1} \in [u, v] \forall y_0 \leq y_1 \leq \ldots \leq y_{r-1} \in [u, v]$$
$$\left(\bigwedge_{i=0}^{r-1} |x_i - y_i| < 2^{-\omega(k)} \to |F(x_0, x_1, \ldots x_{r-1}) - F(y_0, y_1, \ldots y_{r-1})| < 2^{-k} \right). \tag{3.4}$$

If $F(x_k, x_{k+1}, \ldots, x_{k+r-1}) = 0$ for every k, and moreover, the equation $F(x, x,$

$\dots, x) = 0$ *has a unique root with a "modulus of uniqueness"*[1] η, *i.e.*

$$\forall k \forall x, y \in [u, v] \\ \left((|F(x, x, \dots x)| < 2^{-\eta(k)} \wedge |F(y, y, \dots y)| < 2^{-\eta(k)}) \rightarrow |x - y| < 2^{-k} \right), \tag{3.5}$$

then

$$\forall k \forall m > \phi(k)(|x_m - x_{\phi(k)}| \leq 2^{-k}),$$

where $\phi(k) = 2(r-1)(v-u)2^{\omega(\eta(k))}$.

In Section 3.2 we give an example for an application of this result to a theorem in fixed point theory.

The main subject of this paper is to treat a generalization of this result of Matiyasevich where the sequences do not need to be monotone. They need to satisfy the inequality $x_{n+1} \leq (1 + a_n)x_n + b_n$, where (a_n) and (b_n) are non-negative and bounded in sum. We would call such sequences *almost monotone*.

The convergence of sequences satisfying this inequality, introduced by Qihou in [94], finds wide use in fixed point theory in recent papers ([66, 94, 96] among others) to treat Krasnoselski-Mann iterations of asymptotically quasi-nonexpansive mappings with error terms. The form of the inequality reflects the most current iterative schemes used to treat asymptotically non-expansive functions (introduced in [101]) via the (a_n) term and also allows for computational errors in the evaluation of the schemes ([114]) via the (b_n) term.

Error terms are also interesting for recursively-defined sequences, as e.g. computations with computers often introduce errors which can be arbitrarily reduced by increasing the computational precision but never completely eliminated. One would be interested whether a computation of a recursive sequence can start at a low precision and be subsequently refined to achieve the correct final result.

It turns out that it can, provided that the condition on the sequence being almost monotone can be preserved.

The main theorem to be proved in Section 3.3 gives an explicit rate for the convergence of almost monotone recursively-defined sequences, for which moduli of the kind (3.4) and (3.5) can be found. The result also allows computational error, and is uniform in the sense that it only reflects the sequence and recursive definition through a selection of parameters, and is thus applicable to the full range of functions that satisfy the same parameters.

3.2 Application of Matiyasevich's result to fixed point theory

In [38] Hillam proved the following generalization of Krasnoselski's Theorem on the real line:

[1] this term was introduced in [56] in a much more general context. Special forms such moduli also occur in numerical analysis (notably in approximation theory) under the name of strong unicity

Theorem 3.2 *Let $f : [u, v] \to [u, v]$ be a function that satisfies a Lipschitz condition with constant L. Let x_0 in $[u, v]$ be arbitrary and define $x_{n+1} = (1 - \lambda)x_n + \lambda f(x_n)$ where $\lambda = 1/(L+1)$. If (x_n) denotes the resulting sequence, then (x_n) converges monotonically to a point z in $[u, v]$ where $f(z) = z$.*

Hillam proves this statement using three cases:

- $\exists n.f(x_n) = x_n$*:* since $\forall m > n(x_m = x_n = f(x_n))$, the sequence converges to x_n. In the treatment that follows we will allocate this case to one of the others;
- $f(x_0) > x_0$*:* by the continuity of f there exists a fixed point between x_0 and v and then the sequence increases monotonically and is bounded from above by that fixed point[2], therefore by (3.1) it converges;
- $f(x_0) < x_0$ is analogous to the previous.

After that with a simple triangle inequality he proves that the point to which the sequence converges is a fixed point of f.

Suppose that in addition to the conditions in Theorem 3.2 we know that the mapping has a unique fixed point with modulus of uniqueness η. Then, using Matiyasevich's result, we can formulate the following theorem:

Theorem 3.3 *Let $f : [u, v] \to [u, v]$ be a function that satisfies a Lipschitz condition with constant L. Let f have a unique fixed point within $[u, v]$ with a modulus of uniqueness η. Let x_0 in $[u, v]$ be arbitrary and define $x_{n+1} = (1 - \lambda)x_n + \lambda f(x_n)$ where $\lambda = 1/(L + 1)$. Then the following is true:*

$$\forall k \left(|x_{\phi(k)} - f(x_{\phi(k)})| \leq (L + 1)2^{-k} \wedge \forall m > \phi(k)(|x_m - x_{\phi(k)}| \leq 2^{-k}) \right),$$

(i.e. the sequence converges with rate of convergence ϕ and its limit is a fixed point of f) where $\phi(k) = 2(v - u)2^{\eta(k+\lceil \log_2(L+1) \rceil)}$.

Proof. Let $F(y_1, y_2) = (1 - \lambda)y_1 + \lambda f(y_1) - y_2$. Since $\lambda L \leq 1$ this function has a Lipschitz constant 1 and thus a modulus of continuity $\omega_F(k) = k$.

$F(y, y) = \lambda(f(y) - y)$ and thus we can infer that the solution to $F(y, y) = 0$ has a modulus of uniqueness $\eta_F(k) = \eta\left(k + \lceil \log_2(L + 1) \rceil\right)$.

By Matiyasevich's theorem any monotone sequence within $[u, v]$ defined recursively by F converges with rate $\phi(k)$.

Suppose $f(x_0) \geq x_0$. By Hillam's proof either $x_0 < x_1 < \ldots < p$ where p is a the least fixed point of f greater than x_0, or there exists an n, such that $x_0 < \ldots < x_n = \ldots = p$. In either case (x_n) is monotonically increasing and bounded from above by p. Alternatively, if $f(x_0) \leq x_0$, by the same reasoning $(u + v - x_n)$ is monotonically increasing and bounded from above by $u + v - p$.

In both cases the sequence is monotonically increasing and bounded within $[u, v]$, hence Matiyasevich's result applies and therefore

$$\forall k \forall m > \phi(k) \left(|x_m - x_{\phi(k)}| \leq 2^{-k} \right).$$

[2] see the details in [38]

It remains to show that the point it converges to is a fixed point. Let k be arbitrary and $n = \phi(k)$:

$$|x_n - f(x_n)| \leq |x_n - x_{n+1}| + |(1-\lambda)(x_n - f(x_n))|,$$

thus

$$|x_n - f(x_n)| \leq \frac{1}{\lambda}|x_n - x_{n+1}| \leq (L+1)2^{-k}.$$

□

3.3 Almost monotone recursive sequences

We will start with an investigation into some of the properties of almost monotone sequences:

Lemma 3.1 *Let (x_n) be a sequence of non-negative real numbers such that*

$$x_{n+1} \leq (1 + a_n)x_n + b_n \tag{3.6}$$

where $0 \leq a_n, b_n$.

Then for any m and n

$$x_{n+m} \leq (x_n + \sum_{j=0}^{m-1} b_{n+j}) \cdot e^{\sum_{j=0}^{m-1} a_{n+j}}, \tag{3.7}$$

and if additionally $\sum_{i=0}^{\infty} a_i \leq A \in \mathbb{R}$ and $\sum_{i=0}^{\infty} b_i \leq B \in \mathbb{R}$, then

$$\forall n(0 \leq x_n \leq (x_0 + B)e^A). \tag{3.8}$$

Proof. By induction we can show for any $n, m \in \mathbb{N}$

$$x_{n+m} \leq x_n \cdot \prod_{j=0}^{m-1}(1 + a_{n+j}) + \sum_{i=0}^{m-1} b_{n+i} \cdot \prod_{j=i+1}^{m-1}(1 + a_{n+j})$$

and also (by the arithmetic-geometric mean inequality)

$$\prod_{j=0}^{m-1}(1 + a_{n+j}) \leq (1 + \frac{\sum_{j=0}^{m-1} a_{n+j}}{m})^m \leq e^{\sum_{j=0}^{m-1} a_{n+j}}.$$

Combining them yields (3.7), and (3.8) is an instance of this inequality with $n \equiv 0$. □

In this main result, we generalize Matiyasevich's result by extending the class of sequences to almost monotone ones and introducing computational error. Note that, even though we define the error sequence (c_n) separately, it will usually be reflected in the parameters (a_n) and (b_n) as well.

Theorem 3.4 *Let (x_n) be a sequence of non-negative real numbers such that*

$$x_{n+1} \leq (1 + a_n)x_n + b_n$$

and

$$|F(x_n, x_{n+1}, \ldots, x_{n+r-1})| \leq c_n,$$

where $(a_n), (b_n), (c_n)$ are sequences that satisfy

$$0 \leq a_n, \sum_{i=0}^{\infty} a_i \leq A \in \mathbb{R}, \forall k \forall m > \alpha(k)(a_m < 2^{-k}),$$

$$0 \leq b_n, \sum_{i=0}^{\infty} b_i \leq B \in \mathbb{R}, \forall k \forall m > \beta(k)(b_m < 2^{-k}).$$

$$0 \leq c_n, \forall k \forall m > \gamma(k)(c_m < 2^{-k})$$

for some A, α, B, β and γ.

Let $d = (x_0 + B)e^A$ and F be uniformly continuous on $[0, d]$ with modulus ω, i.e.

$$\begin{gathered} \forall k \forall x_0, x_1, \ldots, x_{r-1} \in [0, d] \forall y_0, y_1, \ldots y_{r-1} \in [0, d] \\ \left(\bigwedge_{i=0}^{r-1} |x_i - y_i| < 2^{-\omega(k)} \rightarrow |F(x_0, x_1, \ldots x_{r-1}) - F(y_0, y_1, \ldots y_{r-1})| < 2^{-k} \right) \end{gathered} \tag{3.9}$$

and have a unique solution of $F(x, x, \ldots, x) = 0$ within $[0, d]$ with uniform modulus of uniqueness η, i.e.

$$\begin{gathered} \forall k \forall x, y \in [0, d] \\ \left((|F(x, x, \ldots x)| < 2^{-\eta(k)} \wedge |F(y, y, \ldots y)| < 2^{-\eta(k)}) \rightarrow |x - y| < 2^{-k} \right). \end{gathered} \tag{3.10}$$

Then

$$\forall k \forall m \geq \phi(k) \left(|x_{\phi(k)} - x_m| < 2^{-k} \right), \tag{3.11}$$

where

$$\begin{gathered} \phi(k) = p(r-1) + \max(\alpha(q), \beta(q), \gamma(\eta(k+1)+1)) \\ q = \theta(k) + 3 + \lceil \log_2(d+1)r \rceil \\ p = \lfloor d \cdot 2^{\theta(k)} \rfloor + 1 \\ \theta(k) = \max(k, \omega(\eta(k+1)+1)) \end{gathered}$$

Proof. Lemma 3.1 ensures that all members of the sequence (x_n) lie within $[0, d]$, thus we can safely use the moduli ω and η and freely substitute d as an upper bound for any x_n.

Using α and β we can make sure that, from a certain point onwards, any growth of the sequence (x_n) in a group of r consecutive elements is sufficiently restricted. For any $i \geq \max(\alpha(q), \beta(q))$ (using Lemma 3.1, $r < 2^q$, and $e^x \leq 1 + 2x$ for $x \leq 1$) we have:

$$\begin{aligned} x_{i+j} - x_i &\leq (x_i + j2^{-q})e^{j2^{-q}} - x_i \leq (x_i + j2^{-q})(1 + j2^{-q+1}) - x_i \\ &\leq j2^{-q}(2x_i + 1 + j2^{-q+1}) < (d+1)r2^{-q+2} \leq 2^{-\theta(k)-1} \end{aligned} \tag{3.12}$$

for any $j < r$.

We will now show that a significant distance between elements of (x_n) sufficiently far in the sequence has to be repeated in the distance between another pair of elements. Let n, m be natural numbers, $m, n \geq \max(\alpha(q), \beta(q), \gamma(\eta(k+1)+1))$, and suppose $x_{m+(r-1)} \geq x_n + 2^{-k}$. From its definition we know that $\theta(k) \geq k$ and thus (3.12) yields

$$\begin{aligned} x_m - x_n &= x_{m+(r-1)} - x_n - (x_{m+(r-1)} - x_m) \\ &\geq 2^{-k} - 2^{-\theta(k)-1} \geq 2^{-k-1}. \end{aligned}$$

By the uniqueness (3.10) of the root of $F(x, x, \ldots, x) = 0$ we can infer $|F(x_i, x_i, \ldots, x_i)| \geq 2^{-\eta(k+1)}$ where i is either n or m. By the continuity (3.9) of F applied to $F(x_i, x_i, \ldots, x_i)$ and $F(x_i, x_{i+1}, \ldots, x_{i+r-1})$, where

$$\begin{aligned} |F(x_i, x_i, \ldots, x_i) - F(x_i, x_{i+1}, \ldots, x_{i+r-1})| &\geq 2^{-\eta(k+1)} - c_i \\ &\geq 2^{-\eta(k+1)} - 2^{-(\eta(k+1)+1)} \\ &\geq 2^{-(\eta(k+1)+1)} \end{aligned}$$

we know there must exist $j \in \{1, 2, \ldots, r-1\}$ such that $|x_i - x_{i+j}| \geq 2^{-\omega(\eta(k+1)+1)} \geq 2^{-\theta(k)}$. Because of (3.12) the sequence cannot be growing that much between x_i and x_{i+j}, therefore

$$x_i - x_{i+j} \geq 2^{-\theta(k)}. \tag{3.13}$$

Now the distance between the pair $x_m, x_{n+(r-1)}$ has to be at least 2^{-k} (for simplicity we will only write the case $i = n$, the case $i = m$ yields an identical result):

$$\begin{aligned} x_m - x_{n+(r-1)} &\geq (x_m - x_{m+(r-1)}) + (x_{m+(r-1)} - x_n) + (x_n - x_{n+r-1}) \\ &> -2^{-\theta(k)-1} + 2^{-k} + (x_i - x_{i+j} + x_{i+j} - x_{i+r-1}) \\ &> -2^{-\theta(k)-1} + 2^{-k} + 2^{-\theta(k)} - 2^{-\theta(k)-1} \\ &= 2^{-k}. \end{aligned} \tag{3.14}$$

The same distance is maintained. Provided $m - (r-1)$ continues to be greater than or equal to $\max(\alpha(q), \beta(q), \gamma(\eta(k+1)+1))$, this argument can be applied again.

To continue with the main part of the proof, fix an arbitrary k and let $n_0 = \phi(k)$ and $m_0 \in \mathbb{N}$. Suppose $|x_{m_0+n_0} - x_{n_0}| \geq 2^{-k}$. Consider the following cases:

Case 1. $x_{m_0+n_0} \geq x_{n_0} + 2^{-k}$**.** Let $n_{i+1} = n_i + (r-1)$ and $m_{i+1} = m_i - 2(r-1)$. By induction, using (3.14) with $n = n_i, m = n_i + m_i - (r-1)$ as the induction step, we know that at least for $i \leq \lfloor \frac{m_0}{(r-1)} \rfloor$ (since $n_i + m_i$ remains greater than or equal to $\max(\alpha(q), \beta(q), \gamma(\eta(k+1)+1))$)

$$x_{m_i+n_i} \geq x_{n_i} + 2^{-k}.$$

In particular, for $s = \lfloor \frac{m_0}{2(r-1)} \rfloor$, we have $0 \leq m_s < r$ and $x_{n_s+m_s} - x_{n_s} \geq 2^{-k}$, which is a contradiction with (3.12).

Case 2. $x_{m_0+n_0} \leq x_{n_0} - 2^{-k}$**.** Let $n_{i+1} := n_i - (r-1)$ and $m_{i+1} := m_i + 2(r-1)$. Since $n_0 = \max(\alpha(q), \beta(q), \gamma(\eta(k+1)+1)) + p(r-1)$, we can apply (3.14) p times, taking $n = m_i + n_i$ and $m = n_i - (r-1)$. Therefore for any $i \leq p$ we have

$$x_{n_i} \geq x_{m_i+n_i} + 2^{-k},$$

and moreover (using (3.13)), for each iteration there is a distinct index $l_i \in \{n_i - (r-1), n_i + m_i\}$ where we have a significant drop in the values of the sequence, i.e. where

$$x_{l_i} - x_{l_i+j} \geq 2^{-\theta(k)}$$

for some $j < r$. (note that the drops cannot coincide because the points are at least $(r-1)$-apart)

We will prove that these drops accumulate and our choice of p makes this impossible. We will define two additional sequences to measure how big (x_n) could grow, and what difference there is between that and the real (x_n).

Let $y_0 = x_0, y_{n+1} = (1+a_n)y_n + b_n, z_n = y_n - x_n$. We can easily see that $x_n \leq y_n \leq y_{n+1}$ and $z_{n+1} \geq (1+a_n)y_n + b_n - (1+a_n)x_n - b_n \geq z_n$ for all n. Lemma 3.1 can also be used for (y_n) as an instance of a sequence that satisfies (3.6) with the same constants, thus (y_n) (and thereby (z_n)) also lies within $[0, d]$.

For each $i < p$, there exists $j < r$, such that

$$z_{l_i+j} = y_{l_i+j} - x_{l_i+j} \geq y_{l_i} - x_{l_i+j} = z_{l_i} + x_{l_i} - x_{l_i+j} \geq z_{l_i} + 2^{-\theta(k)},$$

and because of the monotonicity of (z_n) and $l_i \in \{n_i - (r-1), n_i + m_i\}$ also

$$\begin{aligned} z_{m_{i+1}+n_{i+1}} - z_{n_{i+1}} &= z_{m_i+n_i+(r-1)} - z_{n_i-(r-1)} \\ &= z_{m_i+n_i} - z_{n_i} + (z_{m_i+n_i+(r-1)} - z_{m_i+n_i}) + (z_{n_i} - z_{n_i-(r-1)}) \\ &\geq z_{m_i+n_i} - z_{n_i} + 2^{-\theta(k)}. \end{aligned}$$

Iterating this argument we arrive at

$$z_{m_p+n_p} \geq z_{m_p+n_p} - z_{n_p} \geq z_{m_0+n_0} - z_{n_0} + p2^{-\theta(k)} \geq p2^{-\theta(k)} > d,$$

which is a contradiction.

In either case the assumption $|x_{n_0} - x_{m_0+n_0}| \geq 2^{-k}$ causes a contradiction with our choice of n_0, therefore (since k and m_0 were arbitrary)

$$\forall k \forall m \geq \phi(k) \left(|x_{\phi(k)} - x_m| < 2^{-k} \right).$$

□

3.4 Conclusions and future work

In this paper we have approached the problem of recovering effective information from ineffective mathematical proofs by using an approach by Matiyasevich. We have given an application of it and a generalization motivated by recent developments in fixed point theory.

As future work within the topic, we are interested in non-trivial applications of the generalized result. On the other hand, by a result of Kohlenbach in [56], the main prerequisite of the treatments presented here, a modulus of uniqueness, can be extracted under very general conditions even from highly ineffective proofs of the uniqueness of the root. We are interested in finding applications of either the original theorem of Matiyasevich or the generalized result presented here, where finding the modulus is non-trivial, but can be achieved using the theorem from [56].

Chapter 4

Bounds on iterations of asymptotically quasi-nonexpansive mappings

This chapter is a reprint of [66] *"Bounds on Iterations of Asymptotically Quasi-Nonexpansive Mappings"*, coauthored with Ulrich Kohlenbach, in Proceedings of the international conference on Fixed Point Theory and Applications, Valencia 2003, pp. 143-172, Yokohama Publishers 2004.

Abstract

This paper establishes explicit quantitative bounds on the computation of approximate fixed points of asymptotically (quasi-)nonexpansive mappings f by means of iterative processes. Here $f : C \to C$ is a self-mapping of a convex subset $C \subseteq X$ of a uniformly convex normed space X. We consider general Krasnoselski-Mann iterations with and without error terms. As a consequence of our quantitative analysis we also get new qualitative results which show that the assumption on the existence of fixed points of f can be replaced by the existence of approximate fixed points only. We explain how the existence of effective uniform bounds in this context can be inferred already a-priorily by a logical metatheorem recently proved by the first author. Our bounds were in fact found with the help of the general logical machinery behind the proof of this metatheorem. The proofs we present here are, however, completely selfcontained and do not require any tools from logic.

4.1 Introduction

This paper is part of a series of papers which apply tools from mathematical logic to metric fixed point theory ([62, 61, 63, 64] and – for the logical background – [67, 65]). These applications are concerned with both quantitative as well as qualitative aspects of the asymptotic regularity of various iterations of nonexpansive and other mappings in hyperbolic and normed spaces. More specifically, we are interested in effective rates of convergence which are uniform w.r.t. many of the parameters involved. Recently ([67]), the first author proved general logical metatheorems which a-priorily guarantee the existence of such uniform bounds if the convergence statement proved has a certain logical form, and the proof can be carried out in a certain (rather flexible) formal

setting. The proofs of these metatheorems are constructive and allow one to actually extract effective bounds from a given ineffective convergence proof. In this paper we apply this methodology to Krasnoselski-Mann iterations of asymptotically quasi-nonexpansive mappings in uniformly convex normed spaces. We first show how this context fits within the scope of the metatheorems from [67] and then actually construct uniform effective bounds in the main part of this paper which is selfcontained and does not rely on any prerequisites from logic.

In the following, let $(X, \|\cdot\|)$ be a uniformly convex (real) normed linear space and $C \subseteq X$ a nonempty convex subset.

The class of asymptotically nonexpansive mappings $f : C \to C$ was introduced in [26]:

Definition 4.1 *$f : C \to C$ is said to be asymptotically nonexpansive with sequence $(k_n) \in [0,\infty)^{\mathbb{N}}$ if $\lim_{n\to\infty} k_n = 0$ and*

$$\|f^n(x) - f^n(y)\| \leq (1 + k_n)\|x - y\|,\ \forall n \in \mathbb{N}, \forall x, y \in C.$$

Definition 4.2 ([101]) *$f : C \to C$ is said to be uniformly λ-Lipschitzian ($\lambda > 0$) if*

$$\|f^n(x) - f^n(y)\| \leq \lambda\|x - y\|,\ \forall n \in \mathbb{N}, \forall x, y \in C.$$

In the following we use the notation $Fix(f) := \{p \in C : f(p) = p\}$. The concept of quasi-nonexpansive functions was introduced in [18] (based on a similar concept due to [16, 17]):

Definition 4.3 *$f : C \to C$ is quasi-nonexpansive if*

$$\|f(x) - p\| \leq \|x - p\|,\ \forall x \in C, \forall p \in Fix(f).$$

Finally, combining the notions of asymptotically nonexpansive mappings and quasi-nonexpansive mappings we obtain the concept of asymptotically quasi-non-
expansive mappings first studied in [107] and [87] and more recently in [94, 95, 96]:

Definition 4.4 *$f : C \to C$ is asymptotically quasi-nonexpansive with sequence $k_n \in [0,\infty)^{\mathbb{N}}$ if $\lim_{n\to\infty} k_n = 0$ and*

$$\|f^n(x) - p\| \leq (1 + k_n)\|x - p\|,\ \forall n \in \mathbb{N}, \forall x \in X, \forall p \in Fix(f).$$

In the context of asymptotically (quasi-)nonexpansive mappings $f : C \to C$ the so-called Krasnoselski-Mann iteration is defined as follows

$$x_0 := x \in C,\ x_{n+1} := (1 - \alpha_n)x_n + \alpha_n f^n(x_n),$$

where $(\alpha_n) \in [0, 1]^{\mathbb{N}}$.

We will also consider Krasnoselski-Mann iterations with error terms

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n, \tag{4.1}$$

where $\alpha_n, \beta_n, \gamma_n \in [0,1]$ with $\alpha_n + \beta_n + \gamma_n = 1$ and $u_n \in C$ for all $n \in \mathbb{N}$ (this types of error terms was first considered in [114]).

In this paper we study **uniform quantitative versions** as well as **qualitative improvements** of the following theorem which itself seems to be new (though kind of implicit in the literature, see below):

Theorem 4.1 *Let $(X, \|\cdot\|)$ be a uniformly convex (real) normed linear space and C be a convex subset of X. Let (k_n) be a sequence in $\mathbb{R}_+$ with $\sum k_n < \infty$. Let $k \in \mathbb{N}$ and $\alpha_n, \beta_n, \gamma_n \in [0,1]$ such that $1/k \leq \beta_n \leq 1-1/k$, $\alpha_n + \beta_n + \gamma_n = 1$ and $\sum \gamma_n < \infty$. Let $f : C \to C$ a uniformly Lipschitz continuous function such that there exists a $p \in Fix(f)$ with*

$$\forall x \in C \forall n \in \mathbb{N}\big(\|f^n(x) - p\| \leq (1 + k_n)\|x - p\|\big).$$

Define

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n,$$

where (u_n) is a bounded sequence in C. Then the following holds:

$$\|x_n - f(x_n)\| \to 0.$$

Corollary 4.1 *Let $(\alpha_n), (\beta_n), (\gamma_n), (k_n), (u_n), k$ as well as $(X, \|\cdot\|), C$ as in theorem 4.1. If $f : C \to C$ is uniformly Lipschitzian and asymptotically quasi-nonexpansive with sequence (k_n) and $Fix(f) \neq \emptyset$, then $\|x_n - f(x_n)\| \to 0$.*

If $f : C \to C$ is asymptotically nonexpansive with a sequence $(k_n) \in \mathbb{R}_+$ such that $\sum k_n < \infty$ then f automatically is uniformly Lipschitz continuous hence corollary 4.1 implies:

Corollary 4.2 *Let $(\alpha_n), (\beta_n), (\gamma_n), (k_n), (u_n), (X, \|\cdot\|), C$ as in theorem 4.1. If $f : C \to C$ is asymptotically nonexpansive with sequence (k_n) and $Fix(f) \neq \emptyset$, then $\|x_n - f(x_n)\| \to 0$.*

Corollary 4.3 *Let $(X, \|\cdot\|)$ be a uniformly convex Banach space, $C \subset X$ a (nonempty) bounded closed convex subset $(\alpha_n) \in [1/k, 1-1/k]^{\mathbb{N}}$ for some $k \in \mathbb{N}$, $f : C \to C$ asymptotically nonexpansive with sequence (k_n) such that $\sum k_n < \infty$ and*

$$x_0 := x \in C,\ x_{n+1} := \alpha_n x_n + (1 - \alpha_n) f^n(x_n).$$

Then $\|x_n - f(x_n)\| \to 0$.

Proof. Corollary 4.3 follows from corollary 4.2 by omitting the error term (i.e. taking an arbitrary sequence (u_n) in C with $\gamma_n = 0$) and using a theorem from [26] stating that asymptotically nonexpansive mappings $f : C \to C$ always have fixed points (under the given assumptions on X, C). □

The proof of theorem 4.1 is ineffective and the conclusion '$\|x_n - f(x_n)\| \to 0$', i.e.

$$\forall l \in \mathbb{N} \exists n \in \mathbb{N} \forall m \in \mathbb{N}(\|x_{n+m} - f(x_{n+m})\| < 2^{-l}) \tag{4.2}$$

has too complicated a logical form as for our metatheorems to guarantee a computable bound on '$\exists n \in \mathbb{N}$', i.e. a computable rate of convergence. Nevertheless, (4.2) is (non-constructively) equivalent to

$$\forall l \in \mathbb{N} \forall g \in \mathbb{N}^{\mathbb{N}} \exists n \in \mathbb{N}(\|x_{n+g(n)} - f(x_{n+g(n)}\| < 2^{-l}) \tag{4.3}$$

which does have the required logical form so that we can extract a computable bound Φ on '$\exists n$' with g as an additional argument of the bound Φ. The transformed version (4.3) of (4.2) is well-known in logic and called the Herbrand normal form of (4.2). Whereas (4.3) trivially follows from (4.2), the proof of the converse is ineffective.[1] Hence an effective bound on '$\exists n$' in (4.3)' does not lead to an effective bound on '$\exists n$' in (4.2) unless the sequence $(\|x_n - f(x_n)\|)$ is nonincreasing (where this follows already from the special case where $g \equiv 0$) which is e.g. the case for nonexpansive functions f.
Actually, a slightly more flexible form of (4.3) still has a the required logical structure[2]

$$\forall l \in \mathbb{N} \forall g \in \mathbb{N}^{\mathbb{N}} \exists n \in \mathbb{N} \forall m \in [n, n + g(n)](\|x_m - f(x_m)\| < 2^{-l})$$

and we will focus on effective bound $\Phi(l, g)$ on this '$\exists n$'.
In practice, it will be mostly the special case where $g \equiv 0$, i.e.

$$\forall l \in \mathbb{N} \exists n \leq \Phi(l, 0)(\|x_n - f(x_n)\| < 2^{-l})$$

which is of relevance. However, this will not always be sufficient. On general logical grounds though, namely the soundness of the so-called monotone functional interpretation on which our metatheorems are based and the fact that a bound on (4.3) realizes the monotone functional interpretation of (the Gödel negative translation of) (4.2), it follows that a bound for general g **is** sufficient for a quantitative analysis of any use of theorem 4.1 in a proof of a $\forall\exists$-consequence. Our bounds seem to be (already in the case of asymptotically nonexpansive mappings) the only known general quantitative results of that kind (see e.g. [4] for a discussion of the lack of quantitative results in this context).

The qualitative improvement of theorem 4.1 which is obtained via our quantitative analysis consists in the possibility to replace in order to show $\|x_n - f(x_n)\| \to 0$ for a given $x \in C$ the assumption

$$\exists p \in Fix(f) \forall y \in C \forall n \in \mathbb{N}\big(\|f^n(y) - p\| \leq (1 + k_n)\|y - p\|\big)$$

by

$$\exists d \in \mathbb{N} \forall \varepsilon > 0 \exists p_\varepsilon \in Fix_\varepsilon(x, d, f) \forall y \in C, n \in \mathbb{N}\big(\|f^n(y) - f^n(p_\varepsilon)\| \leq (1 + k_n)\|y - p_\varepsilon\|\big),$$

where

$$Fix_\varepsilon(x, d, f) := \{p \in C : \|x - p\| \leq d \wedge \|f(p) - p\| \leq \varepsilon\}.$$

[1] Suppose (4.2) fails for $l \in \mathbb{N}$. Then (4.3) fails for the same l if we take $g(n) := \min m(\|x_{n+m} - f(x_{n+m})\| \geq 2^{-l})$.

[2] Here and below we write $m \in [n, n + g(n)]$ for $m \in \mathbb{N} \wedge m \in [n, n + g(n)]$.

This, of course, is of interest mainly for asymptotically nonexpansive mappings where it replaces the assumption that $Fix(f) \neq \emptyset$ by

$$\forall x \in C \exists d \in \mathbb{N} \forall \varepsilon > 0 (Fix_\varepsilon(x, d, f) \neq \emptyset).$$

With the stronger assumption on (k_n) that $\sum((k_n + 1)^r - 1) < \infty$ for some $r > 1$, corollary 4.3 is proved in [97]. For Hilbert spaces X and $r = 2$ corollary 4.3 is already due to [101]. For Banach spaces satisfying Opial's condition ([90]) and $\sum k_n < \infty$, corollary 4.3 follows from [100] (note, however, that Opial's condition is not even satisfied for L_p except for $p = 2$).[3] The result in the literature most close to corollary 4.1 is the main theorem in [96] whose proof technique – together with an argument reminiscent of a lemma in [101] – we actually use to prove theorem 4.1 and corollary 4.1. The theorem in [96] is concerned with the convergence of (x_n) towards a fixed point of f and the assumption of C being compact.[4] Without that assumption but assuming that $Fix(f) \neq \emptyset$ the proof actually yields that

$$\lim_{n\to\infty} \|x_n - f^n(x_n)\| = 0$$

which together with an argument from [101] gives

$$\lim_{n\to\infty} \|x_n - f(x_n)\| = 0.$$

[96] in turn relies on [99] (see also [109]).

4.2 A logical metatheorem with applications in fixed point theory

This section – which is independent from the main part of this paper – requires some background in logic as developed in [67]. In [67], we have defined a formal system $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$ for classical analysis over a uniformly convex normed space $(X, \|\cdot\|)$ (with a modulus of uniform convexity η) and a bounded convex subset $C \subset X$. The system is formulated in the language of functionals of finite type over the types X (for variables ranging over X-elements) and $\mathbb{N}$ by closure of these types under function space formation: with ρ, τ being types, $\rho \to \tau$ is the type of all functions mapping objects of type ρ to objects of type τ. The type C is treated as a subtype of X. The system contains full countable choice (and hence full comprehension over integers) and even full dependent choice. It is well known in logic that such a system allows to formalize most of existing proofs in analysis. Whereas elements of X are treated as 'primitive' objects (so-called 'atoms') real numbers are – as usual – explicitly represented via Cauchy sequences of rationals numbers with fixed rate of convergence. Both equality $=_{\mathbb{R}}$ on $\mathbb{R}$ as well as equality $=_X$ on X are

[3] For some generalizations of the main results of [100] see also [109].

[4] [96] actually considers more general Ishikawa-type iterations. In order to keep the technicalities of our paper down we confine ourselves here to the Krasnoselski-Mann type iterations.

defined notions where $x =_X y :\equiv \|x - y\| =_{\mathbb{R}} 0$. There are some subtleties, though, which have to do with the restricted availability of extensionality of functions w.r.t. $=_X$. These issues, however, are trivial in our applications in this paper as full extensionality of the functions we will consider follows from the continuity assumptions made (see below).

Definition 4.5 *A formula F is called $\forall$-formula (resp. $\exists$-formula) if it has the form $F \equiv \forall \underline{a}^{\underline{\sigma}} F_{qf}(\underline{a})$ (resp. $F \equiv \exists \underline{a}^{\underline{\sigma}} F_{qf}(\underline{a})$) where $\underline{a}^{\underline{\sigma}} = a_1^{\sigma_1}, \dots, a_k^{\sigma_k}$, F_{qf} does not contain any quantifier and the types in σ_i are $\mathbb{N}$ or C.*[5]

Remark 4.1 *The notions of $\forall$-formula and $\exists$-formula (as well as the theorem corresponding to theorem 4.2 below) from [67] allow more general types. For simplicity we formulate above just the special case needed in this paper.*

Every (real) normed space $(X, \|\cdot\|)$ together with a bounded convex subset C of X gives rise to the 'full' model $\mathcal{S}^{\omega,X}$ over X, C of $\mathcal{A}^\omega[X, \|\cdot\|, C]$. If $(X, \|\cdot\|)$ is uniformly convex and $\eta : \mathbb{N} \to \mathbb{N}$ a modulus of uniform convexity[6] than this model will be a model of $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$. We say that a sentence $A \in \mathcal{L}(\mathcal{A}^\omega[X, \|\cdot\|, C, \eta])$ holds in $(X, \|\cdot\|)$ and C if it holds in this model (see [67] for details on all this).

Definition 4.6 *For functionals x^ρ, y^ρ of type $\rho = \mathbb{N} \to \mathbb{N}$ we define $x \leq_\rho y$ by*

$$x \leq_\rho y :\equiv \forall z^{\mathbb{N}}(x(z) \leq_{\mathbb{N}} y(z)).$$

Theorem 4.2 ([67]) *Let η be a constant of type $\mathbb{N} \to \mathbb{N}$, $\sigma, \rho = \mathbb{N} \to \mathbb{N}$ and $\tau = C, \mathbb{N} \to C$ or $C \to C$. s is a closed term of type $\sigma \to \rho$ and $B_\forall, C_\exists$ are $\forall$- resp. $\exists$-formulas.*
If the sentence

$$\forall x^\sigma \forall y \leq_\rho s(x) \forall z^\tau \big(\forall u^{\mathbb{N}} B_\forall(x, y, z, u) \to \exists v^{\mathbb{N}} C_\exists(x, y, z, v)\big)$$

is provable in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$, then one can extract a computable functional[7] *$\Phi : \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ such that*

$$\forall y \leq_\rho s(x) \forall z^\tau \big[\forall u \leq \Phi(x, b, \eta)\, B_\forall(x, y, z, u) \to \exists v \leq \Phi(x, b, \eta)\, C_\exists(x, y, z, v)\big]$$

holds in any non-trivial (real) uniformly convex normed linear space $(X, \|\cdot\|)$ with convexity modulus η and any non-empty b-bounded convex subset $C \subset X$.

Instead of single variables x, y, z, u, v we may also have finite tuples of variables $\underline{x}, \underline{y}, \underline{z}, \underline{u}, \underline{v}$ as long as the elements of the respective tuples satisfy the same type restrictions as x, y, z, u, v.
Moreover, instead of a single premise of the form '$\forall u^{\mathbb{N}} B_\forall(x, y, z, u)$' we may have a finite conjunction of such premises.

[5] Recall from [67] that the type 'C' is a defined type where – using the notation from [67] – '$\forall x^C A$' and '$\exists x^C A$' stand for '$\forall x^X(\chi_C(x) =_{\mathbb{N}} 0 \to A)$' and '$\exists x^X(\chi_C(x) =_{\mathbb{N}} 0 \wedge A)$', where χ_C represents the characteristic function of C in X.

[6] I.e. $\forall x, y \in X, k \in \mathbb{N}(\|x\|, \|y\| \leq 1 \wedge \left\|\frac{x+y}{2}\right\| \geq 1 - 2^{-\eta(k)} \to \|x - y\| \leq 2^{-k})$.

[7] In the sense type-2 computability theory, i.e. Turing computability w.r.t. oracle Turing machines.

Using the so-called standard representation of compact Polish spaces like $[0,1]^{\mathbb{N}}$ (with the product metric) theorem 4.2 implies the following corollary (see [67]):

Corollary 4.4 *Let $B_\forall, C_\exists$ be $\forall$- resp. $\exists$-formulas and $\varphi : \mathbb{N} \to \mathbb{N}$ be a (primitive recursive function).*
If the sentence

$$\forall n^{\mathbb{N}}, g^{\mathbb{N}\to\mathbb{N}}, (a_k) \in [0,\varphi(n)]^{\mathbb{N}}, x^C, (u_k)^{\mathbb{N}\to C}, f^{C\to C}$$
$$\big(\forall w^{\mathbb{N}} B_\forall(w) \to \exists v^{\mathbb{N}} C_\exists(v)\big)$$

is provable in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$, then one can extract a computable functional $\Phi(n, g, b, \eta)$ such that

$$\forall n, b \in \mathbb{N}, \alpha, \eta \in \mathbb{N}^{\mathbb{N}}, (a_k) \in [0,\varphi(n)]^{\mathbb{N}}, x \in C, (u_k) \in C^{\mathbb{N}}, f : C \to C$$
$$\big(\forall w \le \Phi(n,g,b,\eta)\, B_\forall(w) \to \exists v \le \Phi(n,g,b,\eta)\, C_\exists(v)\big)$$

holds in any non-trivial (real) uniformly convex normed linear space $(X, \|\cdot\|)$ with convexity modulus η and any non-empty b-bounded convex subset $C \subset X$.

Instead of single variables $n, g, (a_n)$ we may also have finite tuples of each of these variables.
Moreover, instead of a single premise of the form '$\forall w^{\mathbb{N}} B_\forall(w)$' we may have a finite conjunction of such premises.

A crucial feature in the above corollary is that the bound $\Phi(n, \alpha, b, \eta)$ does not depend on $(a_k), x, (u_k)$ or f at all and on X and C only via η and b.

Theorem 4.1 can be proved in $\mathcal{A}^\omega[X, \|\cdot\|, C, \eta]$ and even in a weak fragment thereof (as the proof of the quantitative strengthened form of theorem 4.1 given below shows, neither dependent choice DC nor countable choice is needed). Problems in connection with the restricted availability of extensionality in $\mathcal{A}^\omega[X, \|\cdot,\|, C, \eta]$ (see [67]) do not apply here since the assumption on f being (even uniformly) Lipschitz continuous implies the extensionality (see the detailed discussion in the case of nonexpansive functions given in [67]). Hence corollary 4.4 is applicable and guarantees a-priorily a strong uniform effective version of theorem 4.1 in the sense explained in the introduction.

Remark 4.2 (for logicians) *There is a minor problem having to do with the fact that our formal system proves only weak extensionality of the constant χ_C representing the characteristic function of C. As a consequence of this we have to make sure that the condition (we discuss things for notational simplicity here only for the special case of constant sequences) $\alpha + \beta + \gamma = 1$ becomes provable which can be achieved by replacing*

$$(*)\ \forall \alpha, \beta, \gamma \in [0,1](\alpha + \beta + \gamma = 1 \to A(\alpha, \beta, \gamma))$$

officially by

$$(**)\forall \alpha, \beta \in [0,1]\, A(\alpha, \min(1-\alpha, \beta), 1 - \alpha - \min(1-\alpha, \beta)).$$

In the following, though, we will continue to write $(*)$ instead of $(**)$ for better readability.

The assumptions on $\alpha_n, \beta_n, \gamma_n, k_n$ all become $\forall$-formulas once we express '$\sum k_n < \infty$' and '$\sum \gamma_n < \infty$' explicitly with bounds $K, E \in \mathbb{N}$, i.e.

$$A_1 :\equiv$$
$$\forall n \in \mathbb{N}(\alpha_n + \beta_n + \gamma_n =_{\mathbb{R}} 1 \wedge \tfrac{1}{k} \leq_{\mathbb{R}} \beta_n \leq_{\mathbb{R}} 1 - \tfrac{1}{k} \wedge \sum_{i=0}^{n} k_i \leq_{\mathbb{R}} K \wedge \sum_{i=0}^{n} \gamma_i \leq_{\mathbb{R}} E).$$

The existential quantifier '$\exists p \in C$' in the premise

$$\exists p \in C\big(f(p) =_X p \wedge \forall x \in C \forall i \in \mathbb{N}\big(\|f^i(x) - p\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\|\big)\big).$$

can be moved out as a universal quantifier in front of the whole implication leaving back the $\forall$-premise

$$A_2 :\equiv f(p) =_X p \wedge \forall x \in C \forall i \in \mathbb{N}\big(\|f^i(x) - p\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\|\big).$$

Finally, we have the condition on f being uniformly Lipschitz continuous which becomes an $\forall$-formula once stated with a Lipschitz constant $\lambda \in \mathbb{N}$

$$A_3 :\equiv \forall n \in \mathbb{N} \forall x, y \in C\big(\|f^n(x) - f^n(y)\| \leq_{\mathbb{R}} \lambda \cdot \|x - y\|\big).$$

Hence in total, theorem 4.1 can be reformulated as

$$\forall \lambda, k, l, K, E \in \mathbb{N}, g \in \mathbb{N}^{\mathbb{N}}, (k_n) \in [0, K]^{\mathbb{N}}, (\alpha_n), (\beta_n), (\gamma_n) \in [0, 1]^{\mathbb{N}}, x^C, p^C$$
$$\forall (u_n)^{\mathbb{N} \to C}, f^{C \to C}\big(A_1 \wedge A_2 \wedge A_3 \to \exists n \forall m \in [n, n + g(n)](\|x_m - f(x_m)\| <_{\mathbb{R}} 2^{-l})\big).$$

Since the conclusion is (relative to $\mathcal{A}^{\omega}[X, \|\cdot\|, C, \eta]$ equivalent to) an $\exists$-formula and the premise is a conjunction of $\forall$-formulas, we can apply corollary 4.4 to get an effective bound $\Phi(\lambda, k, l, K, E, g, b, \eta)$ on '$\exists n$', i.e.

$$\exists n \leq \Phi(\lambda, k, l, K, E, g, b, \eta) \forall m \in [n, n + g(n)](\|x_m - f(x_m)\| <_{\mathbb{R}} 2^{-l}), \quad (4.4)$$

that does not depend on $(\alpha_n), (\beta_n), (\gamma_n), x, p, f, (k_n), (u_n)$ but only on λ, k, l, K, E, g as well as a bound[8] b on C and the modulus η.

Corollary 4.4 not only provides an effective bound for the conclusion but also allows one to to replace $\forall$-premises by approximate versions thereof. We are here only interested in one of the premises, namely A_2, which can be also written as

$$\forall m \in \mathbb{N}\Big((\|f(p) - p\| \leq_{\mathbb{R}} 2^{-m}) \wedge \forall x \in C \forall i \in \mathbb{N}\big(\|f^i(x) - f^i(p)\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\|\big)\Big), \quad (4.5)$$

where

$$(\|f(p) - p\| \leq_{\mathbb{R}} 2^{-m}) \wedge \forall x \in C \forall i \in \mathbb{N}\big(\|f^i(x) - f^i(p)\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\|\big)$$

itself is a $\forall$-formula. By the corollary we get an effective bound $\Psi := \Psi(\lambda, k, l, K, E, g, b, \eta)$ on '$\forall m$' such that in order to obtain the conclusion (4.4) we can replace (4.5) by

$$\forall m \leq \Psi\Big((\|f(p) - p\| \leq_{\mathbb{R}} 2^{-m}) \wedge \forall x \in C \forall i \in \mathbb{N}\big(\|f^i(x) - f^i(p)\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\|\big)\Big)$$

[8] This requirement will be weakened below.

and hence – since Ψ does not depend on p – by

$$\begin{cases} \forall m \in \mathbb{N} \exists p \in C \Big((\|f(p) - p\| \leq_{\mathbb{R}} 2^{-m}) \wedge \forall x \in C \forall i \in \mathbb{N} \\ \qquad\qquad \big(\|f^i(x) - f^i(p)\| \leq_{\mathbb{R}} (1 + k_i)\|x - p\| \big) \Big). \end{cases}$$

So in effect we have replaced the assumption on f having real fixed points by the weaker assumption on f having approximate fixed points.

The proof of theorem 4.2 in [67] (and hence that of corollary 4.4) is constructive and provides algorithm for actually extracting a bound Φ from the proof of theorem 4.1 together with a proof verifying the bound which – moreover – only uses the existence of ε-fixed points of f. The latter will be shown to always exist for asymptotically nonexpansive mappings by a completely elementary argument, while the existence of real fixed points requires the completeness of X and closedness of C and is based on the non-trivial convex intersection property of uniformly convex Banach spaces, see [26] and also [27]. All this will be carried out in the reminder of this paper. The explicit extraction of the bounds will, furthermore, show that the assumption on C being bounded (needed in the conclusion of the application of theorem 4.2 and hence corollary 4.4) can be replaced by the assumption that (u_n) is bounded (as in theorem 4.1) and that there exists a $d \in \mathbb{N}$ such that within the d-neighbourhood of $x \in C$ there are approximate fixed points $p_\varepsilon \in C$ of f for any $\varepsilon > 0$ (which is trivially satisfied if $Fix(f) \neq \emptyset$). Hence we indeed get in the end a uniform quantitative version of the 'original' theorem 4.1.

4.3 Some helpful lemmata

Lemma 4.1 *Let (a_n) be a sequence in $\mathbb{R}_+$ with $a_{n+1} \leq a_n$ for all n. Then*

$$\forall \varepsilon > 0 \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \max_{i < \lfloor a_0/\varepsilon \rfloor} g^i(0) \left(a_n - a_{g(n)} \leq \varepsilon \right)$$

Proof. The inequality can fail in at most $\lfloor a_0/\varepsilon \rfloor - 1$ steps of applying g, thus it has to be true for at least the one remaining. □

Lemma 4.2 (Quantitative version of a lemma by Qihou, [95]) *Let (a_n), $(b_n), (c_n)$ be sequences in $\mathbb{R}_+$, $A \in \mathbb{Q}_+^*, B, C \in \mathbb{Q}_+$, such that $a_{n+1} \leq (1+b_n)a_n + c_n$; $a_0 \leq A$; $\sum b_n \leq B$; $\sum c_n \leq C$. Then the following hold:*

1. *$(A + C)e^B$ is an upper bound on a_n.*
2. *Let*

$$\Phi(A, B, C, g, \varepsilon) := \max_{i < \lfloor (4BD + 4C + D)/\varepsilon \rfloor} g^i(0),$$

where $D = (A + C)e^D$. Then

$$\forall \varepsilon \in (0, 1] \forall g \in \mathbb{N} \to \mathbb{N} \exists n \leq \Phi(A, B, C, g, \varepsilon) \left(g(n) > n \to |a_{g(n)} - a_n| \leq \varepsilon \right). \tag{4.6}$$

3. Let

$$\Psi(A,B,C,g,\varepsilon) = \Phi(A,B,C,g,\varepsilon/3).$$

Then

$$\begin{aligned} &\forall \varepsilon \in (0,1] \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \Psi(A,B,C,g,\varepsilon) \\ &\forall i,j \left(g(n) \geq j > i \geq n \to |a_j - a_i| \leq \varepsilon \right). \end{aligned} \tag{4.7}$$

Proof. 1: By induction on m one shows

$$a_{n+m} \leq a_n \cdot \prod_{j=0}^{m-1} (1+b_{n+j}) + \sum_{i=0}^{m-1} c_{n+i} \cdot \prod_{j=i+1}^{m-1} (1+b_{n+j})$$

and also (by the arithmetic-geometric mean inequality)

$$\prod_{j=0}^{m-1} (1+b_{n+j}) \leq (1 + \frac{\sum_{j=0}^{m-1} b_{n+j}}{m})^m < e^{\sum_{j=0}^{m-1} b_{n+j}}$$

and combined

$$a_{n+m} \leq (a_n + \sum_{j=0}^{m-1} c_{n+j}) \cdot e^{\sum_{j=0}^{m-1} b_{n+j}}, \tag{4.8}$$

$$a_m \leq (A+C) \cdot e^B$$

for all $m \in \mathbb{N}$.

2: Consider (a_n^*) in which $a_0^* = a_0$ and $a_{n+1}^* = (1+b_n)a_n^* + c_n$. Note $D \geq a_{n+1}^* \geq a_{n+1}^* - a_{n+1} \geq (1+b_n)(a_n^* - a_n) \geq a_n^* - a_n$. Build the two series

$$E_n = 4(BD + C - \sum_{i \leq n} (b_i D + c_i))$$

and

$$D_n = D - (a_n^* - a_n).$$

Their sum satisfies the conditions of Lemma 4.1, therefore there exists $n \leq \Phi(A,B,C,g,\varepsilon)$, such that $E_n - E_{g(n)} \leq \varepsilon$ and $D_n - D_{g(n)} \leq \varepsilon$, but this means

$$\sum_{i=n}^{g(n)} b_i D + \sum_{i=n}^{g(n)} c_i \leq \frac{\varepsilon}{4}.$$

Then (using $e^x \leq 1 + 2x$ for $0 \leq x < 1$)

$$a_j - a_i < (a_i + \frac{\varepsilon}{4}) e^{\frac{\varepsilon}{4D}} - a_i \leq$$

$$(a_i + \frac{\varepsilon}{4})(1 + \frac{\varepsilon}{2D}) - a_i \leq$$

$$\frac{\varepsilon D}{2D} + \frac{\varepsilon}{4} + \frac{\varepsilon^2}{8D} < \varepsilon$$

for all i, j, such that $n \leq i < j \leq g(n)$. That is, if the series grows (i.e. $a_{g(n)} > a_n$), it will satisfy the inequality. If it decreases, then

$$a_n - a_{g(n)} \leq a_n - a_{g(n)} + a^*_{g(n)} - a^*_n = D_n - D_{g(n)} \leq \varepsilon.$$

3: By the proof of the previous point, there is an $n \leq \Psi(A, B, C, g, \varepsilon)$, for which we have $|a_{g(n)} - a_n| \leq \varepsilon/3$ and also $\forall i, j (g(n) \geq j > i \geq n \wedge a_j > a_i \to a_j - a_i \leq \varepsilon/3)$. Therefore, for any i with $g(n) \geq i \geq n$ we have $a_{g(n)} - \varepsilon/3 \leq a_i \leq a_n + \varepsilon/3$ and hence $\forall i, j \in [n, g(n)](a_i \leq a_n + \varepsilon/3 \leq a_{g(n)} + 2\varepsilon/3 \leq a_j + \varepsilon)$, from which the needed inequality follows directly. □

Lemma 4.3 *Let $D : \mathbb{N} \to \mathbb{Q}^*_+, B, C : \mathbb{N} \to \mathbb{Q}_+$, and for all q, let $(a_n)^q, (b_n)^q$, $(c_n)^q$ be sequences in $\mathbb{R}_+$, such that $a^q_{n+1} \leq (1 + b^q_n)a^q_n + c^q_n$; $a^q_n \leq D(q)$; $\sum b^q_i \leq B(q)$; $\sum c^q_i \leq C(q)$ for all $n, q \in \mathbb{N}$. Then:*

1. *Let*

$$\Phi(D, B, C, g, \varepsilon, m) = \max_{i < \lfloor \frac{1}{\varepsilon} \rfloor^{\mathsf{P}}_{q=0}{}^{m-1}(4B(q)D(q)+4C(q)+D(q))\rfloor} g^i(0).$$

Then

$$\forall \varepsilon \in (0, 1] \forall m \in \mathbb{N} \forall g \in \mathbb{N} \to \mathbb{N} \exists n \leq \Phi(D, B, C, g, \varepsilon, m)$$
$$\forall q < m \left(g(n) > n \to |a^q_{g(n)} - a^q_n| \leq \varepsilon \right).$$

2. *Let $\Psi(D, B, C, g, \varepsilon, m) = \Phi(D, B, C, g, \varepsilon/3, m)$. Then*

$$\forall \varepsilon \in (0, 1] \forall m \in \mathbb{N} \forall g \in \mathbb{N} \to \mathbb{N} \exists n \leq \Psi(D, B, C, g, \varepsilon, m)$$
$$\forall q < m \forall i, j \left(g(n) \geq j > i \geq n \to |a^q_j - a^q_i| \leq \varepsilon \right).$$

Proof. As in the previous proof, we can represent every (a^q_n) by two series (D^q_n) and (E^q_n) in a common sum, to which we can apply Lemma 4.1. Then we can carry on the rest of the proof for the individual sequences. □

Definition 4.7 (Clarkson, [11]) *A* modulus of uniform convexity *of a uniformly convex space $(X, \|\cdot\|)$ is a mapping $\eta : (0, 2] \to (0, 1]$, such that for all $x, y \in X$, $\varepsilon \in (0, 2]$*

$$\|x\|, \|y\| \leq 1 \wedge \|x - y\| \geq \varepsilon \to \left\| \frac{x + y}{2} \right\| \leq 1 - \eta(\varepsilon).$$

Lemma 4.4 (Groetsch, [32]) *Let $(X, \|\cdot\|)$ be uniformly convex with modulus η. If $\|x\|, \|y\| \leq 1$ and $\|x - y\| \geq \varepsilon > 0$, then*

$$\|\lambda x + (1 - \lambda)y\| \leq 1 - 2\lambda(1 - \lambda)\eta(\varepsilon), 0 \leq \lambda \leq 1.$$

The next lemma is an adaptation (and improvement) of a lemma from [101] to our situation, i.e. Mann iterations with error term instead of Ishikawa iterations without error term as considered by Schu:

Lemma 4.5 *Let X be a normed linear space, $C \subseteq X$ a convex subset of X, $f : C \to C$ uniformly l-Lipschitzian, and (x_n) be a Krasnoselski-Mann iteration starting from $x \in C$ with error vector (u_n) where $\|u_n - x_n\|$ is bounded by u for all $n \in \mathbb{N}$.*

Then if $\|x_n - f^n(x_n)\| \leq \varepsilon_n$ and $\|x_{n+1} - f^{n+1}(x_{n+1})\| \leq \varepsilon_{n+1}$, then $\|x_{n+1} - f(x_{n+1})\| \leq \varepsilon_{n+1} + (\varepsilon_n + \gamma_n u)(l + l^2)$.

Proof.

$$\begin{aligned}
\|x_{n+1} - f(x_{n+1})\| &\leq \|x_{n+1} - f^{n+1}(x_{n+1})\| + \|f^{n+1}(x_{n+1}) - f(x_{n+1})\| \\
&\leq \varepsilon_{n+1} + l\|f^n(x_{n+1}) - x_{n+1}\| \\
&\leq \varepsilon_{n+1} + l\|f^n(x_{n+1}) - \alpha_n x_n - \beta_n f^n(x_n) - \gamma_n u_n\| \\
&\leq \varepsilon_{n+1} + l\|f^n(x_{n+1}) - f^n(x_n)\| + \\
&\quad + l(\alpha_n + \gamma_n)\|f^n(x_n) - x_n\| + l\gamma_n\|u_n - x_n\| \\
&\leq \varepsilon_{n+1} + l\varepsilon_n + l^2\|x_{n+1} - x_n\| + \gamma_n u l \\
&\leq \varepsilon_{n+1} + l\varepsilon_n + l^2\|\beta_n f^n(x_n) - (\beta_n + \gamma_n)x_n + \gamma_n u_n\| + \\
&\quad + \gamma_n u l \\
&\leq \varepsilon_{n+1} + l\varepsilon_n + l^2\beta_n\|f^n(x_n) - x_n\| + \gamma_n u(l + l^2) \\
&\leq \varepsilon_{n+1} + (\varepsilon_n + \gamma_n u)(l + l^2).
\end{aligned}$$

□

In [26] it is shown – using that reflexive and hence a-fortiori uniformly convex Banach spaces have the so-called 'convex intersection property CIP' – that asymptotically nonexpansive selfmappings of bounded closed convex subsets $C \subset X$ have fixed points. Our quantitative results reduce the need of fixed points to that of approximate fixed points. For the latter we now give a fully elementary proof which does not need CIP (nor the completeness/closedness of X/C):

Lemma 4.6 *Let $(X, \|\cdot\|)$ be a uniformly convex space with modulus η, and $C \subseteq X$ be nonempty, convex and bounded. Let $f : C \to C$ be asymptotically non-expansive with sequence (k_n).*

Then $\mathrm{Fix}_\varepsilon(f) := \{x \in C : \|f(x) - x\| \leq \varepsilon\} \neq \emptyset, \forall \varepsilon > 0$.

Proof. Let $y \in C$. Consider

$$\rho_0 := \inf\left\{\rho \in \mathbb{R}_+ : \exists x \in C \exists k \in \mathbb{N} \forall i > k. \|f^i(y) - x\| \leq \rho\right\}$$

Since C is bounded, the set is non-empty and ρ_0 exists. We also have $\rho_0 \geq 0$ and

$$\forall \delta > 0 \exists x \in C \exists k \in \mathbb{N} \forall i > k. \|f^i(y) - x\| \leq \rho_0 + \delta/2. \tag{4.9}$$

Case 1. $\rho_0 > 0$:

Let $\varepsilon \in (0,4]$ and choose $\delta \in (0,1]$ such that

$$\eta\left(\frac{\varepsilon}{2(\rho_0+1)}\right) > 1 - \frac{\rho_0-\delta}{\rho_0+\delta}.$$

By (4.9), there is an $x_\delta \in C$, such that

$$\exists k \in \mathbb{N} \forall i > k. \|f^i(y) - x_\delta\| \leq \rho_0 + \delta/2. \tag{4.10}$$

Assume that

$$\forall k \in \mathbb{N} \exists n > k. \|f^n(x_\delta) - x_\delta\| \geq \varepsilon/2. \tag{4.11}$$

Let n be large enough that (using (4.11))

$$(1+k_n)(\rho_0+\delta/2) \leq \rho_0 + \delta \wedge \|f^n(x_\delta) - x_\delta\| \geq \varepsilon/2, \tag{4.12}$$

and $m \geq n$ be large enough that (using (4.10))

$$\|f^k(y) - x_\delta\| \leq \rho_0 + \delta/2$$

for all $k \geq m-n$. Then

$$\|f^n(x_\delta) - f^k(y)\| \leq (1+k_n)\|x_\delta - f^{k-n}(y)\| \leq \rho_0 + \delta \tag{4.13}$$

and

$$\|x_\delta - f^k(y)\| \leq \rho_0 + \delta/2 \leq \rho_0 + \delta \tag{4.14}$$

for all $k \geq m$.

(4.12), (4.13) and (4.14) yield by uniform convexity and $\delta \leq 1$

$$\left\|\frac{x_\delta - f^n(x_\delta)}{2} - f^k(y)\right\| \leq \left(1 - \eta\left(\frac{\varepsilon}{2(\rho_0+1)}\right)\right)(\rho_0+\delta) < \rho_0 - \delta$$

for all $k \geq m$, which contradicts the minimality of ρ_0.

Hence (4.11) is false, i.e.

$$\exists k \forall n \geq k. \|f^n(x_\delta) - x_\delta\| < \varepsilon/2,$$

which implies that there exists a k, such that

$$\|f^{k+1}(x_\delta) - x_\delta\| < \varepsilon/2 \text{ and } \|f^{k+2}(x_\delta) - x_\delta\| < \varepsilon/2$$

and hence $\|f(x) - x\| < \varepsilon$ for $x := f^{k+1}(x_\delta)$.

Since $\varepsilon \in (0;4]$ was arbitrary, this implies $\mathrm{Fix}_\varepsilon(f) \neq \emptyset$.

Case 2. $\rho_0 = 0$:

Let $\varepsilon > 0$. Then (4.9) implies

$$\exists x \in C \exists k \in \mathbb{N} \forall i > k. \|f^i(y) - x\| \leq \varepsilon/2$$

and therefore $x_\varepsilon := f^{k+1}(y)$ is an ε-fixed point of f, and again $\mathrm{Fix}_\varepsilon(f) \neq \emptyset$. □

4.4 Main results

Throughout this section, $(X, \|\cdot\|)$ will be a uniformly convex space with modulus of uniform convexity η and C a convex subset of X. f will be a mapping from C to C, $x \in C$ and the series (x_n) will be a Krasnoselski-Mann iteration with error terms (4.1), and $(\alpha_n), (\beta_n), (\gamma_n), (u_n)$ as defined in (4.1).

Theorem 4.3 *Let f be uniformly l-Lipschitzian and*

$$\forall \varepsilon > 0 \exists p_\varepsilon \in C \left(\begin{array}{l} \|f(p_\varepsilon) - p_\varepsilon\| \leq \varepsilon \ \wedge \\ \|p_\varepsilon - x\| \leq d \ \wedge \\ \forall y \in C \forall n \left(\|f^n(y) - f^n(p_\varepsilon)\| \leq (1 + k_n)\|y - p_\varepsilon\| \right) \end{array} \right) \tag{4.15}$$

where $d \in \mathbb{Q}_+^*, k_n \in \mathbb{R}_+$ and also $\sum_{n=0}^{\infty} k_n \leq K \in \mathbb{Q}_+$.

Let $1/k \leq \beta_n \leq 1 - 1/k$ for some $k \in \mathbb{N}$, $\sum_{n=0}^{\infty} \gamma_n \leq E \in \mathbb{Q}_+$, and (u_n) be bounded with $\|u_n - x\| \leq u \in \mathbb{Q}_+$.

Then

$$\forall \delta \in (0,1] \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \Phi \forall m \in [n, n + g(n)] \left(\|x_m - f(x_m)\| \leq \delta \right)$$

where $\Phi = \Phi(K, E, u, k, d, l, \eta, \delta, g)$ and

$$\begin{array}{rcl} \Phi(K, E, u, k, d, l, \eta, \delta, g) & = & h^i(0) \\ h & = & \lambda n.(g(n+1) + n + 1) \\ i & = & \left\lfloor \frac{3(5KD + 6E(U+D) + D)k^2}{\varepsilon \eta(\varepsilon/(D(1+K)))} \right\rfloor \\ D & = & e^K(d + EU) \\ U & = & u + d \\ \varepsilon & = & \delta/(2(1 + l(l+1)(l+2))). \end{array}$$

Proof. Let $\nu \in (0,1) \cap \mathbb{Q}$, p be a p_ε from (4.15), and for the moment assume $\|f(p) - p\| \leq \nu^{n+1}/(n + K)$ is satisfied for all n. Set $U := u + d \geq \|u_n - p\|$. Then we also have $\|f^n(p) - p\| = \|f^{n-1}(f(p)) - f^{n-1}(p) + f^{n-1}(p) - p\| \leq \frac{\nu^{n+1}}{n+K} \sum_{i=0}^{n-1}(1 + k_i) \leq \nu^{n+1}$ by the third clause in (4.15), and

$$\begin{array}{l} \|x_{n+1} - p\| = \|\alpha_n x_n + \beta_n f^n(x_n) + \gamma_n u_n - p\| \\ = \|\alpha_n(x_n - p) + \beta_n(f^n(x_n) - f^n(p)) + \gamma_n(u_n - p) + \beta_n(f^n(p) - p)\| \\ \leq \alpha_n\|x_n - p\| + \beta_n\|f^n(x_n) - f^n(p)\| + \gamma_n U + \beta_n \nu^{n+1} \\ \leq \alpha_n\|x_n - p\| + \beta_n(1 + k_n)\|x_n - p\| + \gamma_n U + \nu^{n+1} \\ \leq (1 + k_n)\|x_n - p\| + \gamma_n U + \nu^{n+1}. \end{array} \tag{4.16}$$

By Lemma 4.2 for all $m \in \mathbb{N}$

$$\|x_m - p\| \leq D, \tag{4.17}$$

where $D := e^K \cdot (d + EU + \nu(1 - \nu))$.

For any n, assume $\|x_n - p\| \geq \varepsilon + \nu^{n+1}$ and $\|f^n(x_n) - x_n\| \geq \varepsilon + \nu^{n+1}$. The latter implies

$$\|(x_n - p) - (f^n(x_n) - f^n(p))\| \geq \|x_n - f^n(x_n)\| - \|p + f^n(p)\| \geq \varepsilon.$$

Hence by Lemma 4.4, using $k_n \leq K$, and (4.17),

$$\left\|(1-\beta_n)\frac{x_n-p}{(1+k_n)\|x_n-p\|}+\beta_n\frac{f^n(x_n)-f^n(p)}{(1+k_n)\|x_n-p\|}\right\| \leq$$

$$1-2\beta_n(1-\beta_n)\cdot\eta\left(\frac{\varepsilon}{(1+K)D}\right). \tag{4.18}$$

Thus

$$\begin{aligned}
&\|x_{n+1}-p\| = \|\alpha_n x_n+\beta_n f^n(x_n)+\gamma_n u_n-p\| \\
&= \|(1-\beta_n-\gamma_n)(x_n-p)+\beta_n(f^n(x_n)-f^n(p)-p+f^n(p))+\gamma_n(u_n-p)\| \\
&\leq \|(1-\beta_n)(x_n-p)+\beta_n(f^n(x_n)-f^n(p))\|+\gamma_n\|u_n-x_n\|+\nu^{n+1} \\
&\leq ((1+k_n)\|x_n-p\|)\left(1-2\beta_n(1-\beta_n)\eta\left(\tfrac{\varepsilon}{(1+K)D}\right)\right) \\
&\quad +\gamma_n(U+D)+\nu^{n+1} \\
&\leq \|x_n-p\|+k_nD+\gamma_n(U+D)+\nu^{n+1}-\varepsilon\cdot 2\tfrac{1}{k^2}\eta\left(\tfrac{\varepsilon}{(1+K)D}\right)
\end{aligned}$$

but $\|x_n-p\| \leq \|x_{n+1}-p\|+|\|x_n-p\|-\|x_{n+1}-p\||$, therefore (4.19) implies

$$0 \leq |\|x_n-p\|-\|x_{n+1}-p\||+k_nD+\gamma_n(U+D)+\nu^{n+1}-2\varepsilon k^{-2}\eta\left(\frac{\varepsilon}{(1+K)D}\right),$$

where the positive additives can be made arbitrarily small by Lemma 4.2, and the negative is a constant greater than 0. Assume we have made the positive sum smaller than this constant for two consecutive members of the series starting at n. By contradiction we will have for both $i=n$ and $i=n+1$

$$\|x_i-p\| < \varepsilon+\nu^{i+1} \text{ or } \|f^i(x_i)-x_i\| < \varepsilon+\nu^{i+1}. \tag{4.19}$$

Consider the following cases:
Case 1. $\|x_{n+1}-p\| < \varepsilon+\nu^{n+2}$.
Here we have

$$\begin{aligned}
\|f(x_{n+1})-x_{n+1}\| &\leq \|f(x_{n+1})-f(p)\|+\|p-x_{n+1}\|+\|p-f(p)\| \\
&\leq (1+l)\|x_{n+1}-p\|+\nu^{n+1} \leq (2+l)(\varepsilon+\nu^{n+1}).
\end{aligned}$$

Case 2. $\|x_{n+1}-f^{n+1}(x_{n+1})\| < \varepsilon+\nu^{n+2}$ and $\|x_n-f^n(x_n)\| < \varepsilon+\nu^{n+1}$.
Then, using Lemma 4.5 with $\varepsilon_{n+1}=\varepsilon_n=\varepsilon+\nu^{n+1}$, we have

$$\|x_{n+1}-f(x_{n+1})\| \leq (\varepsilon+\nu^{n+1}+\gamma_n(U+D))(1+l+l^2).$$

Case 3. $\|x_{n+1}-f^{n+1}(x_{n+1})\| < \varepsilon+\nu^{n+2}$ and $\|x_n-p\| < \varepsilon+\nu^{n+1}$.
In this case we have (reasoning as in (4.20))

$$\|x_n-f^n(x_n)\| \leq (2+l)(\varepsilon+\nu^{n+1})$$

and again using Lemma 4.5

$$\|x_{n+1}-f(x_{n+1})\| \leq (\varepsilon+\nu^{n+1}+\gamma_n(U+D))(1+l(l+1)(l+2)).$$

In either case, if we denote

$$\begin{aligned} p_n &= |\|x_n - p\| - \|x_{n+1} - p\|| \\ q_n &= k_n D + 2\gamma_n(U + D) + 2\nu^{n+1} \end{aligned}$$

and we have

$$\begin{aligned} p_n, q_n &< \varepsilon k^{-2}\eta\left(\tfrac{\varepsilon}{(1+K)D}\right) \\ &\text{and} \\ p_{n+1}, q_{n+1} &< \varepsilon k^{-2}\eta\left(\tfrac{\varepsilon}{(1+K)D}\right), \end{aligned}$$

(note that $|\|x_n - p\| - \|x_{n+1} - p\|| + k_n D + \gamma_n(U + D) + \nu^{n+1} \leq p_n + q_n < 2\varepsilon k^{-2}\eta\left(\frac{\varepsilon}{(1+K)D}\right)$) where $\varepsilon = \delta/(2(1 + l(l + 1)(l + 2)))$, then (using that $q_{n+1} < \varepsilon$)

$$\|x_{n+1} - f(x_{n+1})\| \leq \left(\varepsilon + \frac{q_{n+1}}{2}\right)(1 + l(l + 1)(l + 2)) \leq \delta \tag{4.20}$$

Next, construct the two series

$$\begin{aligned} a_n &= \|x_n - p\| \text{ and} \\ b_n &= KD + 2E(U + D) + \frac{2\nu}{(1 - \nu)} - \sum_{i=0}^{n-1}(k_i D + 2\gamma_i(U + D) + 2\nu^{i+1})) \end{aligned}$$

(note $p_n = a_{n+1} - a_n$, and $q_n = b_{n+1} - b_n$). We know from (4.16) that $a_{n+1} \leq (1 + k_n)a_n + \gamma_n U + \nu^{n+1}$, and $b_{n+1} \leq b_n$, therefore by Lemma 4.3

$$\begin{gathered} \forall k \in \mathbb{N}\forall g : \mathbb{N} \to \mathbb{N}\exists m < \Phi_\nu \forall i, j \\ \left(m - 1 \leq i < j \leq m + g(m) \to |a_j - a_i|, |b_j - b_i| \leq \varepsilon k^{-2}\eta\left(\tfrac{\varepsilon}{(1+K)D}\right)\right), \end{gathered}$$

where

$$\begin{aligned} \Phi_\nu(K, E, u, k, d, l, \eta, \delta, g) &= h^i(0) \\ h &= \lambda n.(g(n + 1) + n + 1) \\ i &= \left\lfloor \frac{3(5KD + 6E(U + D) + 6\nu/(1 - \nu) + D)k^2}{\varepsilon\eta\left(\varepsilon/(D(1 + K))\right)} \right\rfloor \\ D &= e^K(d + EU + \nu/(1 - \nu)) \\ U &= u + d \\ \varepsilon &= \delta/(2(1 + l(l + 1)(l + 2))). \end{aligned}$$

This is enough to ensure (4.19) and hence (4.20) for all $n \in [m, m + g(m)]$ and therefore

$$\begin{gathered} \forall \delta \in (0, 1]\forall g : \mathbb{N} \to \mathbb{N} \\ \exists n \leq \Phi_\nu(K, E, u, k, d, l, \eta, \delta, g)\forall m \in [n, n + g(n)]\left(\|x_m - f(x_m)\| \leq \delta\right). \end{gathered}$$

It only remains to throw away the assumption that $\|f(p) - p\| \leq \nu^{n+1}/(n + K)$ holds for all n. This we can do by simply relaxing it to only the n's for

which the inequality was used in the proof, i.e. for all $n \leq \Phi_\nu$. This is certainly satisfied by $p_{\nu^{\Phi_\nu+1}/(\Phi_\nu+K)}$ using (4.15).

The value of ν was arbitrary within $(0,1) \cap \mathbb{Q}$, thus we can take it arbitrarily small and the bound will get lower at the expense of requiring better approximate fixed points (which we have). Therefore $\Phi = \inf_{\nu\in(0,1)} \Phi_\nu$ will be sufficient for the bound.

Computing the infimum yields the form (4.16). □

Remark 4.3 *Using the argument about the Herbrand normal form (4.3) in Section 1, this theorem and all its corollaries allow us to also conclude*

$$\|f(x_n) - x_n\| \to 0.$$

In particular, theorem 4.3 implies theorem 4.1 from the introduction and is in fact a quantitative strengthening of the latter.

Corollary 4.5 *Let f be uniformly l-Lipschitzian and asymptotically quasi-nonexpansive with sequence (k_n), $\mathrm{Fix}(f) \neq \emptyset$, and also $\sum_{n=0}^{\infty} k_n \leq K \in \mathbb{Q}_+$.*

Let $1/k \leq \beta_n \leq 1 - 1/k$ for some $k \in \mathbb{N}$, $\sum_{n=0}^{\infty} \gamma_n \leq E \in \mathbb{Q}_+$, and (u_n) be bounded with $\|u_n - x\| \leq u \in \mathbb{Q}_+$.

Then

$$\forall \delta \in (0,1] \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \Phi \forall m \in [n, n+g(n)] \left(\|x_m - f(x_m)\| \leq \delta\right)$$

where Φ is as defined in theorem 4.3 with $d \geq \|x - p\|$ for some $p \in \mathrm{Fix}(f)$.

Proof. Direct corollary of the main theorem, where the first and second clauses of (4.15) are satisfied by the existence of real fixed points of f, and the third clause follows from the assumption on f being asymptotically quasi-nonexpansive. □

Corollary 4.6 *If we only need to find a single x_n, which is an approximate fixed point of the function, taking $g(n) \equiv 0$ gives*

$$\forall \delta \in (0,1] \exists n \leq \Phi_1(K, E, u, k, d, l, \eta, \delta) \left(\|x_n - f(x_n)\| \leq \delta\right)$$

where

$$\begin{aligned}
\Phi_1(K, E, u, k, d, l, \eta, \delta) &= \left\lfloor \frac{3(5KD + 6E(U+D) + D)k^2}{\varepsilon\eta\left(\varepsilon/(D(1+K))\right)} \right\rfloor \\
D &= e^K(d + EU) \\
U &= u + d \\
\varepsilon &= \delta/(2(1 + l(l+1)(l+2))).
\end{aligned}$$

Remark 4.4 *If the modulus of uniform convexity of the space can be written in the form $\eta(\varepsilon) = \varepsilon\tilde{\eta}(\varepsilon)$ where $\tilde{\eta}$ is monotone $(0 < \varepsilon_1 \leq \varepsilon_2 \leq 2 \to \tilde{\eta}(\varepsilon_1) \leq \tilde{\eta}(\varepsilon_2))$, the proof of Theorem 4.3 allows to extract a bound with η replaced by $\tilde{\eta}$ (by*

changing $\eta\left(\frac{\varepsilon}{(1+K)D}\right)$ *to* $\eta\left(\frac{\varepsilon}{(1+k_n)\|x_n-p\|}\right)$ *in (4.18) we can replace (4.19) by* $\|x_{n+1}-p\| \leq \|x_n-p\| + k_nD + \gamma_n(U+D) + \nu^{n+1} - \varepsilon \cdot 2\frac{1}{k^2}\tilde{\eta}\left(\frac{\varepsilon}{(1+K)D}\right)$ *and the change carries on through the proof).*

Disregarding the various constants, the ε-dependency of our bounds in the case $g \equiv 0$ is $\varepsilon \cdot \eta(\varepsilon)$.

It is well-known that the Banach spaces L_p with $1 < p < \infty$ are uniformly convex ([11]). For $p \geq 2$, $\frac{\varepsilon^p}{p2^p}$ is a modulus of convexity ([36], see also [62]). Since

$$\frac{\varepsilon^p}{p2^p} = \varepsilon \cdot \tilde{\eta}_p(\varepsilon)$$

where

$$\tilde{\eta}_p(\varepsilon) = \frac{\varepsilon^{p-1}}{p2^p}$$

is monotone, we can apply the previous remark. Hence we get – disregarding again constants – that the ε-dependency of our bounds in the case of L_p ($p \geq 2$) is ε^p.
For the case $X := \mathbb{R}$ with the Euclidean norm, where we can choose $\tilde{\eta}(\varepsilon) := \frac{1}{2}$ (since $\varepsilon/2$ is a modulus of convexity), we have a linear dependency in ε. These results match in quality the bounds obtained in [48, 62, 63] for the case of nonexpansive functions and the usual Krasnoselski-Mann iteration (without error terms). In that case, the deep work in [1] even established a quadratic bound in arbitrary normed spaces for the special case of **constant** $\lambda_n = \lambda \in (0,1)$. For general (λ_n) (satisfying $\lambda_n \in (0, 1-1/k)$ and $\sum \lambda_n = \infty$), the first bounds for Krasnoselski-Mann iterations in arbitrary normed and even hyperbolic spaces were established in [61, 64].

In the case of asymptotically nonexpansive mappings $f : C \to C$ ($C \subset X$ bounded, closed and convex) it is an open problem whether $Fix_\varepsilon(f) \neq \emptyset$, $\forall \varepsilon > 0$, for general (i.e. not uniformly convex) Banach spaces X (see [27], p.135).

Corollary 4.7 *Let f be asymptotically nonexpansive with sequence (k_n), d is such that* $\mathrm{Fix}_\varepsilon(x,d,f) \neq \emptyset$ *for all $\varepsilon > 0$ and also $\sum_{n=0}^{\infty} k_n \leq K \in \mathbb{Q}_+$.*

Let $1/k \leq \beta_n \leq 1 - 1/k$ for some $k \in \mathbb{N}$, $\sum_{n=0}^{\infty} \gamma_n \leq E \in \mathbb{Q}_+$, and (u_n) be bounded with $\|u_n - x\| \leq u \in \mathbb{Q}_+$.

Then

$$\forall \delta \in (0,1] \forall g : \mathbb{N} \to \mathbb{N} \exists n \leq \Phi \forall m \in [n, n+g(n)] \left(\|x_m - f(x_m)\| \leq \delta\right)$$

where Φ is as defined in theorem 4.3 with $l = 1 + K$.

Proof. Direct corollary to the main theorem, using $1 + K \geq 1 + k_n$ for any n as the Lipschitz constant. □

Corollary 4.8 *Let C be a bounded convex subset of X with diameter $d \in \mathbb{Q}_+^*$ and f be asymptotically nonexpansive with sequence (k_n), and also $\sum_{n=0}^{\infty} k_n \leq K \in \mathbb{Q}_+$.*

Let $1/k \leq \beta_n \leq 1 - 1/k$ *for some* $k \in \mathbb{N}$, $\sum_{n=0}^{\infty} \gamma_n \leq E \in \mathbb{Q}_+$.
Then

$$\forall \delta \in (0,1] \forall g : \mathbb{N} \to \mathbb{N}$$
$$\exists n \leq \Phi_2(K, E, k, d, \eta, \delta, g) \forall m \in [n, n + g(n)] \left(\|x_m - f(x_m)\| \leq \delta \right)$$

where

$$\begin{aligned} \Phi_2(K, E, k, d, \eta, \delta, g) &= h^i(0) \\ h &= \lambda n.(g(n+1) + n + 1) \\ i &= \left\lfloor \frac{3(5Kd + 6Ed + d)k^2}{\varepsilon\eta\left(\varepsilon/(d(1+K))\right)} \right\rfloor \\ \varepsilon &= \delta/(2(1 + (K+1)(K+2)(K+3))). \end{aligned}$$

Proof. Using Lemma 4.6 we can fulfill the conditions of the previous corollary, and the boundedness of C allows us to replace all bounds on the distances in the proof with d. □

Concluding remark:

1. With somewhat more complicated bounds our analysis also extends to the case where $f : C \to C$ is instead of being l-uniformly Lipschitzian only ω-uniformly continuous, i.e.

   $$\forall \varepsilon > 0, n \in \mathbb{N}, x, y \in X \big(\|x - y\| < \omega(\varepsilon) \to \|f^n(x) - f^n(y)\| < \varepsilon \big),$$

   where $\omega : \mathbb{R}_+^* \to \mathbb{R}_+^*$ (i.e. ω is what in constructive analysis is called a modulus of uniform continuity for all f^n). In particular, this covers the case of λ-α-uniformly Lipschitzian functions (see [96]).

2. We expect that our analysis can be adapted also to Ishikawa-type iterations. However, this would further complicate the technical details.

Chapter 5

The Basic Feasible Functionals in Computable Analysis

In Section 2.2 we gave a definition of complexity classes of real-valued functions using subrecursive classes of type-2 functionals, including the Basic Feasible Functionals (**BFF**) as a higher-type extension of poly-time computability. We also gave Ko's definition of real function complexity, and specified what it means for a real-valued function to be poly-time in the sense of Ko. This chapter of the thesis will compare the two notions to prove the following theorem:

Theorem 5.1 (5.2) *Let $a, b \in \mathbb{Q}$, $a < b$. A real function $\phi : [a, b] \to \mathbb{R}$ has a sharp CF-representation in **BFF** if and only if it is poly-time computable in the sense of Ko.*

To be poly-time computable in the sense of Ko, a real function ϕ must have a realization in the form of an Oracle Turing Machine (OTM) computing a function $F : (\mathbb{N} \to \mathbb{D}) \to \mathbb{N} \to \mathbb{D}$, which runs in time polynomial in the given precision for all arguments in the domain of ϕ.

To allow that "time polynomial in the given precision" relates to polynomial time computability, all precision arguments in Ko's model are given in unary notation. We will use the equivalent formulation where the precision is taken to be the length of the precision argument.

Another feature of Ko's model for computable analysis is the additional requirement in place for the precision of the representations of real numbers, stating that a representation A of α must satisfy

$$\forall n(\mathrm{prec}(A(n)) = |n| \wedge |A(n) - \alpha| \leq 2^{-|n|}).$$

Since any dyadic number $m.2^{-e}$ is naturally represented as the pair $\langle m, e\rangle$ of integer 'mantissa' and 'exponent', and the function prec can be directly defined as $\mathrm{prec}(m \cdot 2^{-e}) = e$, we will fix the representation of dyadic numbers as the pair $\langle m, e\rangle$. Moreover, since the functions representing real numbers explicitly specify the exponent, we will use this equivalent definition of Ko-computable real numbers and functions:

Definition 5.1 *A function $A : \mathbb{N} \to \mathbb{Z}$ is a Ko-representation of $\alpha \in \mathbb{R}$, if $\forall n.(|A(n) - 2^{|n|}\alpha| \leq 1)$.*

Since dyadic numbers of precision n are represented by numbers of size n plus the number of digits of the integer part of the number, for every closed interval $[a, b]$ with rational endpoints the growth of the representations is polynomially bounded, i.e. there exists a polynomial $p(n) = n + |\max(|a|_{\mathbb{Q}}, |b|_{\mathbb{Q}})|$, such that $|A(n)| \leq p(|n|)$ for all n and for any representation of any real number in $[a, b]$.

The next lemma proves that **BFF** can replace the Oracle Turing Machine running in polynomial time in the definition of poly-time computability according to Ko.

Lemma 5.1 *Let $a, b \in \mathbb{Q}$, $a < b$, and $\phi : [a, b] \to \mathbb{R}$. There exists an Oracle Turing Machine M, which computes a Ko-representation F of ϕ and runs in time polynomial to the precision, if and only if there exists a Ko-representation $G : (\mathbb{N} \to \mathbb{Z}) \to \mathbb{N} \to \mathbb{Z}$ of ϕ in **BFF**. Moreover, whenever A is a Ko-representation of a number $\alpha \in [a, b]$, $F(A) = G(A)$.*

Proof($\to$). Let $B : (\mathbb{N} \to \mathbb{Z}) \to \mathbb{N} \to \mathbb{Z}$ be defined as:

$$B(A, n) := \begin{cases} A(n), & \text{if } P(A, n) = 0 \\ 2^{|n|-(m-1)}c, & \text{otherwise} , \end{cases}$$

where

$$\begin{aligned} P(A, n) &:= \begin{cases} 0, & \text{if } \forall p \leq |n| \, (\lceil 2^p a \rceil - 1 \leq A(2^p) \leq \lfloor 2^p b \rfloor + 1 \wedge \\ & \quad \forall m < p(|A(2^m) - 2^{m-p} A(2^p)| < 2)) \\ 1, & \text{otherwise} \end{cases} \\ m &:= \mu q < |n| \, [P(A, 2^q) = 1] \\ c &:= \max(\min(\lceil 2^{|m|-1} a \rceil, A(2^{m-1})), \lfloor 2^{|m|-1} b \rfloor) \end{aligned}$$

B is a basic feasible functional (because the quantifiers and minimization are sharply bounded) and for every Ko-representation A of a real number in $[a, b]$ $B(A) = A$. Indeed, $\forall n.(|A(n) - 2^{|n|}\alpha| \leq 1)$ implies $\forall m \forall p > m(|A(2^m) - 2^{m-p} A(2^p)| \leq 2)$ and, together with $a \leq \alpha \leq b$, $\forall p(\lceil 2^p a \rceil - 1 \leq A(2^p) \leq \lfloor 2^p b \rfloor + 1)$.

Let A be a sequence of numbers which is not a Ko-representation of a real number in $[a, b]$. Then $\exists n(|A(n) - 2^{|n|}\alpha| > 1 \vee A(n) < 2^{|n|} a - 1 \vee A(n) > 2^{-|n|} b + 1)$. Since $\lfloor x \rfloor \leq x \leq \lceil x \rceil$, this implies that there exists n, such that $\neg P(A, n)$. For every $m \geq n$ we have $B(A, m) := 2^{|m|-|n|} c$, which says that $B(A)$ is a Ko-representation of the number $2^{-(|n|-1)} c$, which is within $[a, b]$.

For any $A : \mathbb{N} \to \mathbb{Z}$, $B(A)$ is a Ko-representation of a real number within $[a, b]$, therefore the Oracle Turing Machine M taking $B(A)$ as input runs in time polynomial to the precision n. We can now use Cook and Kapron's characterization of the **BFF** using Oracle Turing Machines [45]: combining this machine with an OTM implementation of B which will be queried only polynomially many times, we get a machine which runs in time polynomial to the precision n and the size of the argument A, i.e. a basic feasible functional $G := B \circ F$ which matches F for all arguments that are Ko-representations of numbers in $[a, b]$. □

Proof(←). The **BFF** have the Ritchie-Cobham property (see [83]), which means that any **BFF** can be realized by an OTM running in time bounded by $|H(A,n)|$ for some **BFF** H. Let H be this bound for G. The **BFF** are also a hereditarily self-majorized class, which means that there exists a **BFF** H^* such that H^* maj H. From our discussion of the properties of the dyadic numbers we know that there is a common bound for the growth of any Ko-representation of any real number in a closed interval. Let this bound be given as the natural number polynomial p, i.e. $p(|n|) \geq |A(n)|$ for all Ko-representations A of numbers in $[a,b]$. Let $A^* := \lambda n.2^{p(|n|)}$. Since A^* is monotone and everywhere greater than A, A^* maj A. Note also that A^* is a poly-time function.

From the properties of the majorization relation we know that $H^*(A^*)$ maj $H(A)$, and in particular that $H^*(A^*,m) \geq H(A,m)$ for all Ko-representations of numbers in the interval and for all precision requests m, thus $|H^*(A^*,m)|$ is also an upper bound for the running time of the OTM realizing F. But since A^* is a fixed poly-time function, $\lambda m.H^*(A^*,m)$ is a basic feasible functional of type-1, i.e. a poly-time function. Its growth is therefore bounded by some polynomial $q(|m|)$, which proves that $F := G$ can be realized by an Oracle Turing Machine running in time polynomial in the precision for any Ko-representation of any number in $[a,b]$. □

Sharp CF-computability also uses the convention to specify precision requests as the length of the precision argument, but has no restriction on the growth of the number representations, which map precisions to rational approximations.

Let us make sure that we can convert rational to dyadic approximations and vice versa:

Lemma 5.2 *There exist a pair of poly-time functions* $\mathrm{DtoQ} : \mathbb{Z} \to \mathbb{N} \to \mathbb{Q}$ *and* $\mathrm{QtoD} : \mathbb{Q} \to \mathbb{N} \to \mathbb{Z}$*, such that*

$$\mathrm{DtoQ}(m,e) = m \cdot 2^{-|e|}$$

$$\left|\mathrm{QtoD}(q,e) - 2^{|e|}q\right| \leq \frac{1}{2}$$

Proof. Let

$$\mathrm{DtoQ}(m,e) := m \cdot_{\mathbb{Q}} 2^{-|e|}$$

$$\mathrm{QtoD}(q,e) := \left\lfloor 2^{|e|}q +_{\mathbb{Q}} \frac{1}{2} \right\rfloor_{\mathbb{Q}}.$$

The two conversions are poly-time as a composition of poly-time functions. □

We are now ready to prove the main theorem in this chapter.

Theorem 5.2 *Let* $a,b \in \mathbb{Q}$*,* $a < b$*. A real function* $\phi : [a,b] \to \mathbb{R}$ *is poly-time computable in the sense of Ko if and only if it has a sharp CF-representation among the* ***BFF****.*

Proof(→). Let $F : (\mathbb{N} \to \mathbb{Z}) \to \mathbb{N} \to \mathbb{Z}$ be the basic feasible functional obtained by Lemma 5.1. Let $G : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$ be the following functional:

$$G(A) := \lambda n.\mathrm{DtoQ}(F(\lambda m.\mathrm{QtoD}(A(2m), m), 2n), 2n),$$

where DtoQ and QtoD are two conversion functions defined above.

Since every function(al) in this definition is basic feasible, and the class of the basic feasible functionals is closed under composition and functional substitution, G is a basic feasible functional.

Now let us prove that it is a sharp CF-representation of ϕ: Let A be a CF-representation of a number $\alpha \in [a, b]$. We have that for any m, $|A(2m) - \alpha| < 2^{-|m|-1}$ and therefore $|\mathrm{QtoD}(A(2m), m) - 2^{|m|}\alpha| < 1$, which means that $B := \lambda m.\mathrm{QtoD}(A(2m), m)$ is a Ko-representation of α.

From the definition of F we have that $F(B)$ is a Ko-representation of $\phi(\alpha)$. Since $|\mathrm{DtoQ}(F(B, 2n), 2n) - \alpha| \leq 2^{-|2n|} < 2^{-|n|}$, the final application of DtoQ converts it to the CF-representation $G(A)$. □

Proof(←). Let $G : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$ be a CF-representation of a real function $\phi : [a, b] \to \mathbb{R}$. Let $F : (\mathbb{N} \to \mathbb{Z}) \to \mathbb{N} \to \mathbb{Z}$ be defined as:

$$F(B) := \lambda n.\mathrm{QtoD}(G(\lambda m.\mathrm{DtoQ}(B(2m), 2m), 2n), n).$$

Let B be a Ko-representation of a real number $\alpha \in [a, b]$. Reasoning as above, $A := \lambda m.\mathrm{DtoQ}(B(2m), 2m)$ defines a CF-representation of $\alpha \in [a, b]$, we can also see that $G(A)$ is a CF-representation of $\phi(\alpha)$ and therefore $\lambda n.\mathrm{QtoD}(G(A, 2m), m)$ is a Ko-representation of it.

F is a basic feasible functional that maps Ko-representations of the input to Ko-representation of the result. By Lemma 5.1 there exists an Oracle Turing Machine implementing F and running in time polynomial to the precision. □

Intuitively, the forward direction of the proof works because, using the Cook/Kapron characterization, a basic feasible functional has sufficient time to: read its input and convert it to a dyadic number with precision n (given by the length of the real argument), process the input via a black-box copy of the Ko-representation (given by the precision argument), and enough time to translate the output to a rational number (given again by the precision argument). The inverse argument relies on the fact that the BFF CF-representation of the function never generates big numbers in itself, and since the arguments it is given have a polynomially bounded growth, all computations stay polynomially bounded in all three steps of the conversion.

This result was originally observed as part of [75], which discusses complexity and intensionality in an interval framework for analysis (which will also be given in Chapters 6 to 7 of the thesis). We decided to give it more emphasis in a separate chapter, because the result in itself is interesting and gives extra robustness to both Melhorn's complexity notion and Ko's.

Chapter 6

Computable Analysis via Partial Approximation Representations

This chapter is a revised version of part of [75], *"Complexity and Intensionality in a Type-1 Framework for Computable Analysis"*. Ong, L. (ed.), Computer Science Logic: 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005. Proceedings. Lecture Notes in Computer Science, vol. Volume 3634 (2005), pp. 442-461.

6.1 Introduction

This chapter of the thesis presents an approach to computable analysis which corresponds to interval arithmetic supplied with a mechanism for increasing precision. The approach is designed to allow very fast implementations of exact real arithmetic.

One way to achieve good performance for an implementation is to have real functions able to operate on simpler objects rather than the functions that describe real numbers. To make it easy for the user to write programs using these simpler objects, one must have modularity on them, meaning that one should be able to generate compositions of functions in a straightforward manner. This is not possible to do using the representations of rational numbers that are the simple objects in the approach of CF-representations. It is, however, possible if the internal objects are intervals.

The core of the model is one of the equivalent definitions for real function computability given by Grzegorczyk in [34]. Real functions in this model map intervals to intervals and are not suspect to a monotonicity requirement. The original definition is the third alternative described in Section 2.1.

In this chapter we present a definition that is not confined to functions with closed intervals as domains, and is able to define functions like division without requiring an explicit apartness bound. To do this, we must allow the possible interval ends to include $\pm\infty$ in order to be able to specify cases where we have no information about the result. Because of this, we use the term *partial approximations* to denote intervals approximating a real number.

An additional problem that is solved in this definition is the possible failure of the original approach when binary functions are considered. Due to the fact

that it only requires that a certain size of the result is eventually reached, when two arguments combine it may happen that they do not become small intervals at the same time, which means that too elaborate a scheme must be employed to correctly apply binary functions. Instead, we choose to resolve the problem by requiring that the representations of real numbers produce intervals that eventually *stay* smaller than an arbitrary size bound.

Additionally, this chapter gives a modification of the approach that can be used to implement intensional representations of multi-valued functions with the same efficiency.

We give proof of the equivalence of this approach to CF-computability in a construction that is later also applied to the intensional partial approximation representations, and in the following chapter to subrecursive classes to reason about complexity in the approach. The real number library described in Chapter 8 relies on this approach to provide exact real computations with overhead that is low enough to make it possible for the library to be used in cases where hardware floating point is often sufficient.

6.2 Definitions

The basic objects we are going to use contain approximation information and an estimation of the amount of error in this approximation. To be able to define a class of real functions equivalent to the CF-computable ones, a totally indeterminate value has to be permitted (otherwise e.g. division cannot be defined, see [105]). We do this by allowing an infinitely large value for the error.

Let $\mathbb{V}$ be an enumerable dense subset of the real numbers that is closed under addition, substraction and division by 2, and $\mathbb{E}$ be a subset of the positive rational numbers which contains 1 and is closed under multiplication and division by 2, to which the special value ∞ is added. We assume both can be computably encoded and converted from one to the other (using upwards rounding in the case of conversions from $\mathbb{V}$ to $\mathbb{E}$), the basic operations are computable, and the comparison operator respects ∞. Possible choices for $\mathbb{V}$ include $\mathbb{Q}$ and $\mathbb{D}$; $\mathbb{E}$ can, for instance, be defined by adding $+\infty$ to $\mathbb{Q}$, $\mathbb{D}$, the set of numbers 2^e for an integer e, or $m \cdot 2^e$ where m is a small (e.g. $m < 2^{32}$) integer.

Having the distinction between the sets $\mathbb{V}$ and $\mathbb{E}$ is prompted by the need to include ∞, but also closely follows the choice one would often make when an actual implementation is developed as one might prefer a simpler (and thus more efficient) representation of the error information.

Definition 6.1 *A* partial approximation *to a real number α is a pair $\langle v, e\rangle$ of type $\mathbb{V} \times \mathbb{E}$, such that $|v - \alpha| < e$. We will denote the class of partial approximations to α with $\mathbb{A}_\alpha$, and the class of partial approximations to any real with $\mathbb{A}_\mathbb{R} = \cup_{\alpha\in\mathbb{R}}\mathbb{A}_\alpha$. If $a \in \mathbb{A}_\mathbb{R}$ we will use a_v, a_e to denote respectively the value and error in a.*

Definition 6.2 *A* partial approximation representation (p.a.r.) *of a real number α is a function $A : \mathbb{N} \to \mathbb{A}_\alpha$, for which $\forall k \exists n \forall m \geq n((A(m))_e \leq 2^{-k})$.*

If a real number is CF-computable, then it certainly has a computable p.a.r.: if B is a CF-representation of α, then $\lambda n.(B(n), 2^{-n})$ is one of its p.a.r.'s. Conversely, if a is a p.a.r. of α, then

$$\lambda k.(A(\mu n[(A(n))_e \leq 2^{-k}]))_v \tag{6.1}$$

is a valid CF-representation for it.

For real functions, we want to have objects that operate on partial approximations instead of the full representations. They will have to convert approximations to an input to approximations to the result of the application of the function, and also we need to require that the precision of the output approximations gets arbitrarily good as the precision of the input increases. In other words,

Definition 6.3 *A* partial approximation representation *of a partial function $\phi : \mathbb{R} \to \mathbb{R}$ is a partial function $F : \mathbb{A}_\mathbb{R} \to \mathbb{A}_\mathbb{R}$, such that for any choice of $\alpha \in$ dom ϕ and a partial approximation representation A of α, $\lambda n.F(A(n))$ is a partial approximation representation of $\phi(\alpha)$.*

Remark 6.1 *This definition implies $a \in \mathbb{A}_\alpha \to F(a) \in \mathbb{A}_{\phi(\alpha)}$ for $\alpha \in$ dom ϕ.*

6.3 Computability

We have severely restricted the information to which the function object has access; nevertheless, this does not restrict the class of real functions that are computable. The following theorem is a proof of this fact that uses a construction which we will later modify to use in our complexity and intensionality results:

Theorem 6.1 *A partial function $\phi : \mathbb{R} \to \mathbb{R}$ is CF-computable if and only if it has a computable p.a.r.*

Proof(←). If we have a p.a.r. F of a function ϕ, and $\alpha \in$ dom ϕ, then the functional

$$\Phi(B, n) := (F(\langle B(m), 2^{-m} \rangle))_v, \text{ where} \tag{6.2}$$
$$m = \mu p \left[(F(\langle B(p), 2^{-p} \rangle)_e \leq 2^{-n} \right]$$

is total in n for any CF-representation B of α since from Definitions 6.3 and 6.2 the minimization will always stop, and Definition 6.1 together with Remark 6.1 ensures $|\Phi(a, n) - \phi(\alpha)| < 2^{-n}$. □

Proof(→). Fix a CF-representation Φ for ϕ.

For any $a \in \mathbb{A}_\mathbb{R}$ with $a_e < 1$, we can effectively find the largest natural number m with the property $2^m a_e < 1$. If $a_e \geq 1$, we take $m = 0$. Define the function

$$b(n) := 2^{-n} \lfloor 2^n a_v + 1/2 \rfloor. \tag{6.3}$$

For $0 \leq n < m$ we have that if $\alpha \in \mathbb{A}_\alpha$

$$|b(n) - \alpha| \leq |a_v - \alpha| + 2^{-(n+1)} \leq 2^{-m} + 2^{-(n+1)} \leq 2^{-n}$$

In the following we will use the language of exceptions[1]. Given the code of a computable functional Φ, we can construct an equivalent one $\Phi^\dagger$ that honors a new exception x. We can create a function

$$b\lceil m := \lambda n. \begin{cases} b(n), & \text{if } n < m \\ \textbf{raise } x, & \text{otherwise} \end{cases}$$

and then define

$$\Phi^\ddagger(B,n) := \begin{cases} \langle 0, \infty \rangle, & \text{if } n = 0 \\ \textbf{try } \left\langle \Phi^\dagger(B, n-1), 2^{-(n-1)} \right\rangle & \\ \quad \textbf{catch}(x)\ \Phi^\ddagger(B, n-1) & \text{otherwise} \end{cases}, \tag{6.4}$$

($\Phi^\ddagger(b\lceil m, n)$ finds the largest $l \leq n-1$ for which $\Phi(b,l)$ only refers to the first m values in b, or returns a completely undefined value if such an n cannot be found).

We will now prove that the function

$$F(a) := \Phi^\ddagger(b\lceil m, m+1) \tag{6.5}$$

is the required p.a.r. of ϕ. To do this, we need to prove that $G = \lambda n.F(A(n))$ is a p.a.r. of $\phi(\alpha)$ for any p.a.r. A of α.

The first condition, $F(a) \in \mathbb{A}_{\phi(\alpha)}$ for any $a \in \mathbb{A}_\alpha$, follows from the requirement for Φ and the fact that there is a CF-representation for α that starts with $b(0), b(1), \ldots, b(m-1)$.

For the second condition, we need to prove the existence of 2^{-k}-approximations to $\phi(\alpha)$ among $G(n)$ for any k. The sequence defined by

$$c(n) := 2^{-n}\lfloor 2^n\alpha + 1/2 \rfloor$$

is a proper CF-name for α. If α is not a dyadic number, then for an arbitrary n, $|\alpha - c(n)| < 2^{-n-1}$. There exists q depending on n, such that $|\alpha - c(n)| \leq 2^{-n}(1/2 - 2^{-(q-n)})$, and for all partial approximations a with $a_e < 2^{-q}$ we have $2^i|a_v - c(i)| < 1/2$ for all $0 \leq i \leq n$. But this implies that the sequence obtained by (6.3) coincides with c on the first $n+1$ elements.

Now, since Φ would look at finitely many elements of c to produce a value with any precision 2^{-k}, using that count in the procedure described above, we can come up with a q supplying a long enough sequence. Combining this with a requirement that m in (6.5) is sufficient for the target precision, we have $(F(a))_e \leq 2^{-k}$ for all a's with $a_e \leq 2^{-\max(q,k)}$, and since A has only finitely

[1]The reader can refer to a current book on semantics (e.g. [85]) for a proper definition of the concept and its implementation. Essentially the same approach (but explicitly specified and not identified as a case of using exceptions) is used e.g. in [54] and [3] and even in the definition of Kleene's associates [51]. Through the use of exceptions we avoid the tedious explicit construction of the functional $\Phi^\ddagger$ from the code of Φ.

many elements that have a bigger error, this is satisfied for $a = A(n)$ for all n beyond a certain point.

If α is a dyadic number, i.e. $\exists n(c(n) = \alpha)$, then there are only finitely many variations of b that can exists, because they have to coincide after the first $n + 1$ positions. Then there exists a maximum m for the number of lookups Φ can make to any of these b's in order to get a 2^{-k}-precise result. Hence $a_e \leq 2^{-\max(m,k)}$ suffices to get the required precision for $F(a)$. □

6.4 Intensional representations

Intensionality does not work well with the type-1 frameworks, because intensional functions rely on information that is not available in an approximation. If Φ is not extensional, Theorem 6.1 does not hold. More specifically, a partial function given by a p.a.r. is always extensional:

Theorem 6.2 *Let $F : \mathbb{A}_\mathbb{R} \to \mathbb{A}_\mathbb{R}$ and let $\alpha \in \mathbb{R}$ such that for all p.a.r. A of α, $\lambda n.F(A(n))$ is a p.a.r. of some $\beta \in \mathbb{R}$. Then β depends only on α and not on its representation A.*

Proof. Let X and Y be two p.a.r.'s of α. Then

$$Z(n) = \begin{cases} X(\frac{n}{2}), & \text{if } n \text{ is even} \\ Y(\frac{n-1}{2}), & \text{otherwise} \end{cases}$$

is also a representation of α. Then $\lambda n.F(Z(n))$ is a p.a.r. of a real number β and therefore $\lambda n.F(X(n))$ and $\lambda n.F(Y(n))$ are also p.a.r.'s to β as subsequences of $\lambda n.F(Z(n))$. □

Still, intensional functions are interesting for us and we want to find a way to accommodate them. To do this, we have to pass additional information to the functions.

The most straightforward solution is to supply information about the history of the approximation as an argument to the p.a.r., i.e. essentially use Kleene's associate definition [51]. We will not be treating this approach, because the amount of information that has to be passed to the associate in a direct application of Kleene's approach is too big and complexity reasoning would be very difficult if not impossible.

A different approach, carrying less information, is to give the function access to the previous value it has produced, i.e.

Definition 6.4 *A recursion-p.a.r. of a multi-valued function ϕ is a function $F : \mathbb{A}_\mathbb{R} \times \mathbb{A}_\mathbb{R} \to \mathbb{A}_\mathbb{R}$, such that for any choice of $\alpha \in$ dom ϕ and p.a.r. A of α, $\lambda n.F(A(n), F(A(n-1), F(A(n-2), \cdots F(A(0), 0) \cdots)))$ is a p.a.r. of a $\beta \in \phi(\alpha)$.*

Alternatively, one can extract the "history information" in a separate function:

Definition 6.5 *A storage-p.a.r. of a multi-valued function ϕ is a pair of functions $F : \mathbb{N} \to \mathbb{A}_{\mathbb{R}}$ and $H : \mathbb{A}_{\mathbb{R}} \times \mathbb{N} \to \mathbb{N}$, such that for any choice of $\alpha \in$ dom ϕ and p.a.r. A of α, $\lambda n.F(H(A(n), H(A(n-1), H(A(n-2), \cdots H(A(0), 0) \cdots)))$ is a p.a.r. of a $\beta \in \phi(\alpha)$.*

The idea behind this is that the function has access to a memory cell where it can store information about past calls and update at each call. This can be very efficient, especially in practical cases where a couple of bits of external storage[2] can be sufficient.

We will be treating the storage-p.a.r. approach and in the end of the chapter we will show that the two are equivalent.

Theorem 6.3 *A multi-valued function has a CF-representation if and only if it has a storage-p.a.r.*

Proof(←). Given a storage-p.a.r. pair F, H, the function

$$\begin{array}{rcll} \Phi(a,n) & = & F(h(k))_v, \text{ where} & \\ h(m) & = & \begin{cases} 0, & \text{if } m = 0 \\ H(\left\langle a(m-1), 2^{-(m-1)} \right\rangle, h(m-1)), & \text{otherwise} \end{cases} & \\ k & = & \mu m.\, [F(h(m))_e \leq 2^{-n}] & \end{array}$$

is a CF-representation of the function ϕ: h builds a sequence of applications of H which is only lengthened when we move ahead in the approximation, and since the sequence $\left\langle a(i), 2^{-i} \right\rangle_{i \in \mathbb{N}}$ is a p.a.r. to the argument, the storage-p.a.r. of ϕ has to return approximations to one of the possible results, and the minimization for k always terminates. □

Proof(→). We will define H that builds a signed digit representation of the real number and adds more information to it with consecutive calls. We will be storing the signed digit representation as a pair $\langle h_i, h_s \rangle$, where h_i is an integer approximating the number with error 1, and h_s is a string of $\{-1; 0; 1\}$ encoded in base 4. The following function $H : \mathbb{A}_{\mathbb{R}} \to \mathbb{N} \to \mathbb{N}$ implements this:

$$H(a, \langle h_i, h_s \rangle) = \begin{cases} \langle h_i, h_s \rangle, & \text{if } a_e > 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \\ \left\langle \lfloor a_v + \frac{1}{2} \rfloor, 0 \right\rangle, & \text{if } \frac{1}{4} < a_e \leq \frac{1}{2} \wedge \langle h_i, h_s \rangle = 0 \\ \langle g_i, 4g_s + 1 \rangle, & \text{if } a_e \leq 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \wedge \\ & \text{man}(\langle h_i, h_s \rangle) - a_v 2^{\exp(\langle h_i, h_s \rangle)} > \frac{1}{2} \\ \langle g_i, 4g_s + 3 \rangle, & \text{if } a_e \leq 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \wedge \\ & \text{man}(\langle h_i, h_s \rangle) - a_v 2^{\exp(\langle h_i, h_s \rangle)} < -\frac{1}{2} \\ \langle g_i, 4g_s + 2 \rangle, & \text{otherwise}, \end{cases}$$

[2] In practice, F will usually take the current approximation as an additional argument. This argument is not needed for the proofs that follow and does not interfere with them because H can encode it in its result.

where

$$\begin{array}{rcl}
\langle g_i, g_s\rangle & = & H(\langle a_v, 2a_e\rangle, \langle h_i, h_s\rangle) \\
\mathrm{man}(\langle h_i, h_s\rangle) & = & \begin{cases} h_i, & \text{if } h_s = 0 \\ 2\mathrm{man}(\langle h_i, \lfloor \frac{h_s}{4} \rfloor\rangle) - 1, & \text{if } h_s \equiv 1 (\mathrm{mod} 4) \\ 2\mathrm{man}(\langle h_i, \lfloor \frac{h_s}{4} \rfloor\rangle), & \text{if } h_s \equiv 2 (\mathrm{mod} 4) \\ 2\mathrm{man}(\langle h_i, \lfloor \frac{h_s}{4} \rfloor\rangle) + 1, & \text{if } h_s \equiv 3 (\mathrm{mod} 4) \end{cases} \\
\exp(\langle h_i, h_s\rangle) & = & \left\lceil \frac{|h_s|}{2} \right\rceil .
\end{array}$$

We use the same construction as in Theorem 6.1 (changing only the definitions of m and b), to prove that the following is a storage-p.a.r. of ϕ if Φ is its CF-representation and $\Phi^\dagger$ is a version of it that honors a new exception x:

$$F(\langle h_i, h_s\rangle) = \Phi^\ddagger(b\lceil m, m+1)$$

for

$$\begin{array}{rcl}
\Phi^\ddagger(B,n) & = & \begin{cases} \langle 0, \infty\rangle, & \text{if } n = 0 \\ \textbf{try } \left\langle \Phi^\dagger(B, n-1), 2^{-(n-1)} \right\rangle \\ \quad \textbf{catch}(x)\ \Phi^\ddagger(B, n-1) & \text{otherwise} \end{cases} \\
m & = & \exp(\langle h_i, h_s\rangle) \\
b\lceil m & = & \lambda n. \begin{cases} b(n), & \text{if } n \le m \\ \textbf{raise } x, & \text{otherwise} \end{cases} \\
b(n) & = & \mathrm{dya}(\mathrm{man}(g(n)), \exp(g(n))) \\
g(n) & = & \left\langle h_i, \lfloor h_s 4^{n-m} \rfloor \right\rangle .
\end{array}$$

In this b decodes the information stored in h to a unary function which gives correct approximations to the argument up to its m'th value and $\Phi^\ddagger$ computes $\Phi(b, n)$ for the largest $n \le m$ for which this information is sufficient. If the information in $\langle h_i, h_s\rangle$ is not yet sufficient to approximate the integer part of the number, m will be 0 and thus $\Phi^\ddagger$ will return an indeterminate value.

Let A be a p.a.r. of an $\alpha \in$ dom ϕ and h be a shorthand for $h(n) = H(A(n), H(A(n-1), \cdots H(A(0), 0)\cdots))$.

Since A contains approximations to α for any precision, the string built by h is has no limit for its length and encodes a CF-representation of α. Since Φ is a computable CF-representation of ϕ, by passing to it finite parts of this representation of α, we are getting finite parts of the representation of a number $\beta \in \phi(\alpha)$, and the construction of $\Phi^\ddagger$ ensures $F(h(n)) \in \mathbb{A}_\beta$. To get arbitrarily precise approximations to β it suffices to be able to provide arbitrarily long finite parts of the CF-representation of α. We can do this, and all further elements of the transformed sequence will be at least this precise. □

Unlike in Theorem 6.1, where b can be different at consecutive calls to F with different approximations to the number, here the initial part of b does not change and this makes the proof simpler.

Finally, it remains to show that the recursion-p.a.r. approach shares the same properties:

Theorem 6.4 *A multi-valued function ϕ has a recursion-p.a.r. if and only if it has a storage-p.a.r.*

Proof. Let R be a recursion-p.a.r. of ϕ. Then

$$\begin{aligned} H(a,h) &= R(a,h) \\ F(h) &= h \end{aligned}$$

is a storage-p.a.r. of ϕ. Conversely,

$$\begin{aligned} R(a,h) &= \mathrm{hide}(F(H(a,\mathrm{extr}(h))), H(a,\mathrm{extr}(h))) \\ \mathrm{hide}(a,h) &= \begin{cases} \langle h, \infty \rangle, & \text{if } a_e \geq \frac{1}{2} \\ \mathrm{hh}(a,h), & \text{otherwise} \end{cases} \\ \mathrm{hh}(a,h) &= \left\langle 2^{-\mathrm{ll}(a_e)} \left(\lfloor 2^{\mathrm{ll}(a_e)} a_v \rfloor + 2^{-|h|} \left((2^{|h|} - 1) + 2^{-(|h|+1)} h \right) \right), 2a_e \right\rangle \\ \mathrm{extr}(a) &= \begin{cases} a_v, & \text{if } a_e \geq 1 \\ \mathrm{ee}(2^{\mathrm{ll}(a_e)+1} a_v - \lfloor 2^{\mathrm{ll}(a_e)+1} a_v \rfloor), & \text{otherwise} \end{cases} \\ \mathrm{ee}(z) &= 2^{\mathrm{count}(z)+1}(2^{\mathrm{count}(z)} z - (2^{\mathrm{count}(z)} - 1)) \\ \mathrm{ll}(e) &= \begin{cases} 0, & \text{if } e \geq 1 \\ 1 + \mathrm{ll}(2e), & \text{otherwise} \end{cases} \\ \mathrm{count}(z) &= \begin{cases} 0, & \text{if } z < \frac{1}{2} \\ 1 + \mathrm{count}(2z-1), & \text{otherwise} \end{cases} \end{aligned}$$

does the translation in the other direction: R hides the values of h inside the results it returns by truncating a_v to the precision of a_e and adding to it a string of ones as long as the binary representation of h, followed by a zero and h itself. Because a_e is doubled, a is still a partial approximation to the result of the application of the function, and the value of h can be extracted by first removing the truncated a_v, counting the number of consecutive ones in the remainder and then recovering h as the string of this length that follows the separating zero. □

6.5 Partial Approximation Representations in Practice

The partial approximations used in our model directly correspond to intervals of the real line with endpoints in $\mathbb{V} \cup \{\pm\infty\}$. For an easier presentation we made explicit the center and error bound in such an interval, but a practical implementation may choose to use any method of representing intervals that it may see fit.

Moreover, the framework allows arbitrary overestimation in the evaluation of functions on intervals, as long as the condition that given a converging sequence of interval they must return converging sequences. The functions may overestimate arbitrarily within these limits.

For example, the four arithmetic operations on intervals $x = [\underline{x}, \overline{x}]$ and $y = [\underline{y}, \overline{y}]$ may be implemented using the standard method

$$x \odot y = [\min(\underline{x} \odot \underline{y}, \underline{x} \odot \overline{y}, \overline{x} \odot \underline{y}, \overline{x} \odot \overline{y}), \max(\underline{x} \odot \underline{y}, \underline{x} \odot \overline{y}, \overline{x} \odot \underline{y}, \overline{x} \odot \overline{y})]$$

through exact computations using rational numbers. Another approach could use diadic numbers to represent the endpoints of the intervals and rounding

to make the results of the operations dyadic of the same length as the inputs. A third approach may choose to evaluate the result of the application of the operations to the centers only to the precision of the inputs and give an error bound depending on that precision and the continuity behavior of the function.

All these methods will fit in the p.a.r. framework. In the Domain Theoretic framework [19], for example, it would not be obvious how the latter method could be made monotone. The p.a. representations of real functions need not satisfy that requirement.

The p.a.r. approach is modular, meaning that one can compose real functions to build new functions without having to analyze the construction of the building blocks. The resulting objects continue to work on simple approximations to real numbers. In order to use modularity in the model of CF-representations or the TTE model, one must use objects that represent real numbers completely, which is quite inefficient.

The modularity property is very important from a practical point of view, since it gives the possibility to imitate working on complete reals while the actual computations are taking place on the approximations level. For example, the reciprocal square root of a real number may be defined as the function $(\lambda x.\sqrt{x}) \circ (\lambda x.\frac{1}{x})$. If recip implements $\frac{1}{x}$ in the p.a.r. model and sqrt – square root, then simply sqrt $\circ$ recip would be a definition the reciprocal square root in the p.a.r. model, even though none of these representations has access to the complete real number.

Schwichtenberg's approach to analysis with witnesses [102] also has modularity for the objects that carry out the computations, i.e. the objects that convert rational approximations to rational approximations. However, it relies on moduli of continuity every time it needs to extract actual information from the objects. The moduli, in turn, rely on worst-case analysis for the precision that a function may require, which on the average requests much higher precision than can be shown to be actually required by the error propagation analysis done by partial approximation representations.

In some cases the method to first evaluate the required precision using moduli of continuity and then compute an approximation using that is called top-down evaluation, while the error propagation analysis is called bottom-up evaluation. This issue will be further discussed in Chapter 8, where we describe an actual implementation based on the p.a.r. model.

The best feature of this implementation is its possibility to achieve performance close to the very best alternative method while providing certified correctness. The p.a.r. model gives sufficient freedom to use a wide variety of techniques in the same theoretical model.

The p.a.r. approach is sometimes implicitly used by numerical analysts when they want to be certain about a result they have achieved. Using interval arithmetic with varying precision until the end result is precise enough to fit some criteria is an application of the p.a.r. model. In fact, this chapter gives proof that this approach is powerful enough and represents exactly the functions computable via CF-representations, a seemingly unrelated approach.

While the idea of using interval arithmetic with increasing precision may be pretty straightforward, it is not that obvious how discontinuous functions

should be handled. The intensional variations of the partial approximation representations are a well specified approach that could handle many problems arising in practice.

The intensional p.a. representations are given in two variants, so that an implementation can make the best possible use of resources. In many cases the storage-p.a.r. approach may provide the best performance, because some functions would only require two bits to store whether they have made a choice and in which direction it was taken. Alternatively, in cases where the implementation can make use of the previous approximation in computing the next one (as e.g. in computing functions using the Newton method), the recursion-p.a.r. approach would be more useful.

Chapter 7

Complexity of Partial Approximation Representations

This chapter is a revised version of part of [75], *"Complexity and Intensionality in a Type-1 Framework for Computable Analysis"*. Ong, L. (ed.), Computer Science Logic: 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005. Proceedings. Lecture Notes in Computer Science, vol. Volume 3634 (2005), pp. 442-461.

7.1 Introduction

Since the partial approximation representations of real numbers are allowed to approach their limit arbitrarily slow, complexity reasoning based on the class of functions used in the definition of a p.a.r. of a real number or function alone is meaningless because all classes define the same set of numbers and functions, namely all computable ones. For example,

Theorem 7.1 *If a real number has a computable p.a. representation, it has a poly-time p.a. representation.*

Proof. Let A be one of the number's p.a. representations and e be the program code of A.

The poly-time functions can define the minimization normal form of the partial recursive functions, i.e. there exist (see e.g. [89]) poly-time functions $U : \mathbb{N} \to \mathbb{N}$ and $T : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, such that

$$\phi_e(x) \downarrow \Leftrightarrow \exists y(T(e, x, y) = 0)$$

$$\phi_e(x) \simeq U(\mu y[T(e, x, y) = 0].$$

The function

$$B(x) := \begin{cases} U\left(\mu z < |x|\left[T(e, w, z) = 0\right]\right), & \text{if } \exists k, z < |x|\left(T(e, k, z) = 0\right) \\ \langle 0, \infty \rangle, & \text{otherwise,} \end{cases}$$

where

$$w = |x| - \mu k < |x|\left[\exists z < |x|\left(T(e, |x| - k, z) = 0\right)\right],$$

is poly-time, because pairing and sharply bounded quantification and minimization are poly-time computable. It is a p.a.r. of the number represented by A, since if we can eventually find a point n where all subsequent values of A have error smaller than 2^{-k} for any k, after the point 2^m, for m such that $T(e, n, m) = 0$, all values returned by B will return $A(x)$ for some $x \geq n$, i.e. having error smaller than 2^{-k}. □

The same trick works for functions, but we must first impose some requirements on the encodings of $\mathbb{V}$, $\mathbb{E}$ and their pairings in order to be able to perform operations on them. We request that the encodings satisfy the following requirements:

- $|\langle a, b\rangle|$ is polynomial in $\max(|\langle a\rangle_{\mathbb{V}}|, |\langle b\rangle_{\mathbb{E}}|)$
- $\langle a, b\rangle \geq \langle a\rangle_{\mathbb{V}}$ and $\langle a, b\rangle \geq \langle b\rangle_{\mathbb{E}}$
- $\langle 2^{-n}\rangle_{\mathbb{E}} \geq 2^n$
- there exist poly-time functions that convert between the encodings of $\mathbb{V}$ and $\mathbb{E}$, rounding up if a number in $\mathbb{V}$ cannot be represented in $\mathbb{E}$
- multiplication and division by 2 are poly-time (and thus also multiplication by $2^{\pm|k|}$) in both $\mathbb{V}$ and $\mathbb{E}$
- addition and the floor function $\lfloor\cdot\rfloor$ in $\mathbb{V}$ are poly-time
- there exists a function $\mathrm{dya}(n, d)$ that selects a code for the dyadic number $n2^{-d}$, such that whenever a, b, c, d are positive integers, $a \leq c \wedge b \leq d \rightarrow \mathrm{dya}(a, b) \leq \mathrm{dya}(c, d)$
- the absolute value operator on the codes is such that $\langle v\rangle_{\mathbb{V}} \leq \langle |v|\rangle_{\mathbb{V}}$ for any $v \in \mathbb{V}$

In the case $\mathbb{V} = \mathbb{Q}$ and $\mathbb{E} = \mathbb{Q}^+_\infty$ these properties are satisfied e.g. by the Cantor pairing, the encoding of rational numbers q as $\langle n, d\rangle$, such that

$$q = (-1)^n \frac{\lfloor (n+1)/2 \rfloor}{d},$$

and the encoding of ∞ as $\langle 0, 0\rangle$.

Theorem 7.2 *If a partial real function has a computable p.a. representation, it has a poly-time p.a. representation.*

Proof. Let F be one of the function's p.a. representations and e be the program code of F. Let $b(a, n) = \left\langle \mathrm{dya}(\lfloor 2^{|n|} a_v + 1/2 \rfloor, |n|), 2^{-|n|} \right\rangle$. If α is a real number such that $a \in \mathbb{A}_\alpha$ and m is the biggest number such that $2^{-m} > a_e$, then $\forall n < m(b(a, n) \in \mathbb{A}_\alpha)$.

The function

$$G(a) := \begin{cases} U\left(\mu z < |a|\left[T(e, w, z) = 0\right]\right), & \text{if } \exists k < m \exists z < |a| \\ & \quad (T(e, b(a, 2^k), z) = 0) \\ \langle 0, \infty\rangle, & \text{otherwise}, \end{cases}$$

where

$$w = m - 1 - \mu k < m[\exists z < |a|\,(T(e, b(a, 2^{m-1-k}), z) = 0)]$$
$$m = \mu k < |a|\,[2^{-(k+1)} \leq a_e],$$

is poly-time because $m < |a|$ and thus bounded by m means sharply-bounded by a, we can also compute 2^k if $k \leq m$, sharply-bounded minimization and quantification are poly-time operations, and b is a poly-time function.

Following the reasoning in Theorem 6.1, we know that for any p.a.r. A of a number α in the domain of the function, as l grows $\lambda n.b(A(l), n)$ becomes a match for longer and longer initial sections of one of finitely many approximations of α. Because the choices are finitely many, for any $k \in \mathbb{N}$ there exist common bounds m and n, such that $\exists p < m \forall l > n((F(b(A(l), p)))_e < 2^{-k})$. Moreover, there exists a common bound z for the computation code, i.e. $\exists n < m \exists y < z \forall l > n(T(e, b(A(l), p), y) = 0 \wedge (U(y))_e < 2^{-k})$.

Since for every p.a.r. A there exists a point q such that $\forall l > q((A(l))_e < 2^{-\max(m+2,z)})$, $\lambda l.G(A(l))$ is a p.a.r. of the number represented by $\lambda l.F(A(l))$, because $\forall l > q(G(A(l)))_e < 2^{-k}$. □

7.2 Real numbers.

In order to be able to speak about different complexity classes of real numbers, we must make a definition which requests more from our functions in order to avoid the minimization in (6.1). This gives rise to the following definitions and equivalence property:

Definition 7.1 *A* modulus *for a p.a.r. A of a real number α is a function $m : \mathbb{N} \to \mathbb{N}$, such that for all k and all $n > m(k)$, $(A(n))_e \leq 2^{-|k|}$.*

Definition 7.2 *We will say that a real number is p.a.r.-computable in a given class C of computable functions, if there exist both a p.a.r. and a modulus for it in C.*

Theorem 7.3 *A real number has a sharp CF-representation in a subrecursive class C that contains the poly-time functions and is closed under composition if and only if it is p.a.r.-computable in C.*

Proof. If B is a sharp CF-representation of the number, take the p.a.r. $A(n) := \langle B(n), 2^{-|n|} \rangle$ and the modulus $m(n) := n$.

For the other direction $B(k) := (A(m(k)))_v$ is a sharp CF-representation of the number if A and m are, respectively, its p.a.r. and modulus. □

7.3 Type-2 complexity for functions.

Again taking the p.a.r. of a real function we lose all complexity information about that function. To talk about complexity classes, we define a function that can replace the minimization in (6.2) in a way that can also be used for multi-argument functions:

Definition 7.3 *A* modulus *for a p.a.r. F of a partial real function ϕ is a partial functional $M : (\mathbb{N} \to \mathbb{A}_{\mathbb{R}}) \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \longrightarrow \mathbb{N}$, such that for all $\alpha \in$ dom ϕ, p.a.r. A of α, moduli m for A,*

$$\forall k \forall n > M(A, m, k)((F(A(n)))_e \leq 2^{-|k|}).$$

Note that even though the actual function object is a type-1 object, we now introduce a type-2 operation to characterize it. However, some extra flexibility comes from the separation of these two objects: to implement e.g. a feasible real function one does not have to implement a feasible type-2 object, but only needs to prove that it exists. Moreover, if a CF-representation of a function needs extra information to be in a certain class (e.g. division needs evidence that the denominator is non-zero to be primitive recursive), it will in general only be needed for the modulus.

Definition 7.4 *We will say that a real function is p.a.r.-computable in a given class C of computable type-2 functionals, if both a computable p.a.r.*[1] *and its modulus can be found in C.*

Theorem 7.4 *If a function is p.a.r.-computable in a given class C that contains **BFF** and is closed under functional composition and functional substitution for total functions, then it is computable in the same class.*

Proof. For $\phi : \mathbb{R} \to \mathbb{R}$, $\alpha \in$ dom ϕ, F- p.a.r. of ϕ, M-modulus for F, and B - CF-representation of α, take

$$\Phi(B, n) := (F(A(M(A, \lambda p.p, n))))_v$$

where

$$A(p) := \left\langle B(p), 2^{-|p|} \right\rangle.$$

A is a p.a.r. for α with a modulus $\lambda p.p$, and hence from M being a modulus to F, we have $|\Phi(B, n) - \phi(\alpha)| < 2^{-|n|}$. Φ is obtained by composition and functional substitution from basic feasible functionals and F and M, therefore it is in C. □

The other direction is more complicated. First we will verify that p.a.r.-computability coincides with CF-computability, i.e. that, in addition to the p.a.r., a modulus can be found for every computable function:

Theorem 7.5 *If a partial function $\phi : \mathbb{R} \to \mathbb{R}$ is CF-computable, then it is p.a.r.-computable in the class of all partial computable functionals.*

Proof. For this proof we will need to modify the proof of Theorem 6.1. The proof we used in Chapter 6 made a case selection according to a non-computable predicate (whether a number is dyadic or not), which does not let us explicitly construct a modulus.

[1] via the implicit embedding of Type 1 in Type 2

The modified proof will rely on exhausting all possible initial sequences that lead to the current approximation and choosing from them the one that gives the best approximation of the output. With this modification we can be sure that once a given interval results in an approximation with a given precision, all subintervals of it will result in approximations of at least this precision. More precisely, let $\Phi : (\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$ be a CF-representation of the real function ϕ. Let $\alpha \in \text{ dom } \phi$ and fix $a \in \mathbb{A}_\alpha$. Let

$$b_r(n) = \begin{cases} \langle \lfloor 2^n a_v \rfloor, 2^n \rangle, & \text{if } r_n = 0 \\ \langle \lceil 2^n a_v \rceil, 2^n \rangle, & \text{if } r_n = 1 \end{cases}$$

(i.e. given a binary string r, $b_r(n)$ produces dyadic approximations rounded according to the digits in r), let $\Phi^\dagger$ be a version of Φ that honors a new exception x, $b_r\lceil m$ be defined as

$$b_r\lceil m := \lambda n. \begin{cases} b_r(n), & \text{if } n < m \\ \textbf{raise } x, & \text{otherwise} \end{cases}$$

and define the function $\Phi^\ddagger$ as

$$\Phi^\ddagger(B, n) := \begin{cases} \langle 0, \infty \rangle, & \text{if } n = 0 \\ \textbf{try } \left\langle \Phi^\dagger(B, n-1), 2^{-(n-1)} \right\rangle & \\ \quad \textbf{catch}(x)\ \Phi^\ddagger(B, n-1) & , \quad \text{otherwise} \end{cases}$$

($\Phi^\ddagger(b_r\lceil m, n)$ finds the largest $l \leq n-1$ for which $\Phi(b_r, l)$ only refers to the first m values in b_r, or returns a completely undefined value if such an n cannot be found).

We will now prove that the function

$$F(a) := \Phi^\ddagger(b_r\lceil m, m+1),$$

where

$$\begin{aligned} r &= \mu s \left[\forall t < 2^m \left((\Phi^\ddagger(b_s\lceil m, m+1))_e \leq \Phi^\ddagger(b_t\lceil m, m+1))_e \right) \right] \\ m &= \mu k[2^{-(k+2)} < a_e] \end{aligned}$$

is a p.a.r. of ϕ and will then define a modulus for this function.

All $b_r(n)$ for $n < m$ are approximations to α of precision 2^{-n}. Moreover, when we look at the functions b_r that would be generated by a subinterval d of a, we will see that they will contain a match of these and therefore $(F(d))_e \leq (F(a))_e$.

The canonical CF-representation c of α defined as

$$c(n) = 2^{-n} \left\lfloor 2^n \alpha + \frac{1}{2} \right\rfloor$$

will also be matched in its initial components by some b_r. Since c is a valid (though possibly non-computable) CF-representation of α, for every precision k, $\Phi(c)$ would have to request a finite part of c, which will eventually be matched

by a b_r generated by a sufficiently small approximation $a \in \mathbb{A}_\alpha$. Thus F converts approximations to α to approximations to $\phi(\alpha)$.

Let $M(A,m,k) := \mu n.[F(\langle (A(m(n)))_v, 3(A(m(n)))_e\rangle)_e \leq 2^{-k}]$. The minimization halts, because selecting a subsequence of unboundedly decreasing error from the p.a.r. we still have a p.a.r. of the same number and thus any target precision will be reached. Additionally, for any $n \geq M(A,m,k)$ we have that $(A(n))_e \leq (A(M(A,m,k)))_e$ and therefore

$$A(n) \subseteq \langle (A(M(A,m,k)))_v, 3(A(M(A,m,k)))_e\rangle$$

and $(F(A(n)))_e \leq 2^{-k}$. □

To prove the equivalence between p.a.r.-computability and the existence of sharp CF-representations in some restricted type-2 computability classes, we need the higher-type monotonicity we have in the hereditarily self-majorized classes such as the ones described in Section 2.2.2. The arguments to a real function need not be subrecursive or even computable, but still the following lemma shows that we can bound them in a suitable sense:

Lemma 7.1 *Let b be defined as*

$$b(n) := \mathrm{dya}(\lfloor 2^{|n|} a_v + 1/2\rfloor, |n|). \tag{7.1}$$

Then for all $\alpha \leq a_0, a \in \mathbb{A}_\alpha, b$, created by (7.1) for a with $a_e \leq 1$,

$$J(a_0)\ \mathrm{maj}_1\ b$$

where

$$J(a_0) = \lambda n.\mathrm{dya}(1 + \lfloor 2^{|n|} a_0 + 1/2\rfloor, |n|). \tag{7.2}$$

Proof. Since $a_e \leq 1$, we have $|a_v| < a_0 + 1$ and therefore by the properties of the encoding $J(a_0)(n) \geq b(n)$, and also, since when n is increased both the numerator and denominator in (7.2) do not decrease, we have $\forall k \leq n(J(a_0)(n) \geq J(a_0)(k) \geq b(k))$, which means $J(a_0)\ \mathrm{maj}_1\ b$. □

We are now ready to prove the result using another modification of the proof of Theorem 6.1.

Theorem 7.6 *If a partial real function has a sharp CF-representation in a hereditarily self-majorized class of type-2 functionals that contains **BFF** and is closed under composition and functional substitution for total functions, then it is p.a.r.-computable in that class.*

Proof. Let Φ be a sharp CF-representation of the partial real function ϕ.

We will use the proof of Theorem 6.1, substituting the definition (6.3) of b with (7.1) and changing all functions used in that proof to basic feasible versions (making use of the fact that Φ is now a sharp CF-representation of the function). The function F defined as

$$F(a) := \Phi^\ddagger(b\lceil m, 2^{m+1}\rceil)$$

where $\Phi^\dagger$ is a version of Φ that honors the new exception x,

$$\begin{aligned}
\Phi^\ddagger(B,n) &:= \begin{cases} \langle 0,\infty\rangle, & \text{if } n=0 \\ \textbf{try } \left\langle \Phi^\dagger(B,\lfloor \frac{n}{2}\rfloor), 2^{-\left|\lfloor \frac{n}{2}\rfloor\right|}\right\rangle \textbf{ catch}(x)\ \Phi^\ddagger(B,\lfloor \frac{n}{2}\rfloor), & \text{otherwise} \end{cases} \\
b\lceil m &:= \lambda n. \begin{cases} b(n), & \text{if } n<m \\ \textbf{raise } x, & \text{otherwise} \end{cases} \\
b(n) &:= \mathrm{dya}(\lfloor 2^{|n|}a_v + 1/2\rfloor, |n|) \\
m &:= \mu k < |a|\,[2^{-(k+2)} < a_e]
\end{aligned}$$

does the same job as the original definition in Theorem 6.1 and thus the same reasoning tells us that it is a partial approximation representation of ϕ.

All operations used in the generation of F can be done without leaving the class of Φ (this is true because m is bounded by $|a|$ and thus all definitions here can be expressed as composition or functional substitution of function(al)s in the class[2]). Hence F is in the class. We now need to find a modulus for it.

In the class of Φ there exists a functional Ψ that does exactly the same job as Φ, but instead of returning the approximation it gives the largest k to which B was applied. Since the class contains this functional and is hereditarily self-majorized, it also contains a majorizer Ψ^* for it. The modulus for A gives us means to bound the absolute value of the real number described by it, therefore, with the previous lemma, there is a poly-time function $b^* := J(|A(m(0))|+1)$ which majorizes all functions b generated by partial approximations with error less than 1.

Hence $l = \Psi^*(b^*,n) \geq \Psi(b,n)$ for all good b's, in particular for the one (call it b_0) generated by $a_0 = A(m(l))$, which means $\Phi^\dagger(b_0\lceil l, n)$ will not raise an exception, and $F(a_0)$ will give a result with the required precision.

Hence $M(A,m,n) = \max(m(\Psi^*(J(|A(m(0))|+1),n)),n)$ is a modulus for F. □

7.4 Complexity of Intensional Representations

Complexity measures can be introduced similarly to the extensional case, but here we also want to make sure the history information does not grow too quickly:

Definition 7.5 *A* modulus *for a storage-p.a.r. pair F,H of a function ϕ is a pair of functions $M,N : (\mathbb{N}\to\mathbb{A}_\mathbb{R})\times(\mathbb{N}\to\mathbb{N})\times\mathbb{N} \longrightarrow \mathbb{N}$, such that for any p.a.r. A to $\alpha \in$ dom ϕ with modulus m,*

$$\forall k \left(F(H(A(m(2^{|n|})), H(A(m(2^{|n|-1})), \cdots H(A(m(0)),0)\cdots)))_e \leq 2^{-k}\right),$$

where $n = M(A,m,k)$ and

$$\forall n \left(H(A(m(2^{|n|})), H(A(m(2^{|n|-1})), \cdots H(A(m(0)),0)\cdots) \leq N(A,m,n)\right)$$

[2]the recursion on notation $\Phi^\ddagger$ can be made bounded using the same trick that shows $\max_{x<|y|} F(x)$ is in **BFF** for a function argument F (see [13]).

Theorem 7.7 *A multi-valued function has a sharp CF-representation in a hereditarily self-majorized subrecursive class of functionals that includes the **BFF** and is closed under composition and functional substitution for total functions, if and only if it has a storage-p.a.r. and its modulus in the same class.*

Proof(←). This variation of what we did in the previous theorem is in **BFF** if F, H, M and N are in **BFF**:

$$\begin{aligned}
\Phi(a,n) &= F(h(k))_v, \text{ where} \\
h(m) &= \begin{cases} 0, & \text{if } m = 0 \\ H(\left\langle a(2^{|m|-1}), 2^{-(|m|-1)} \right\rangle, h(2^{|m|-1})), & \text{otherwise} \end{cases} \\
k &= M(\lambda p.(a(p), 2^{-|p|}), \lambda p.p, n),
\end{aligned}$$

because the recursion on notation h is bounded by $N(\lambda p.(a(p), 2^{-|p|}), \lambda p.p, n)$. □

Proof(→). All the constructions used in Theorem 6.3 can be done in **BFF** using a modification similar to the previous theorem. The function H, defined as (not changed from Theorem 6.3):

$$H(a, \langle h_i, h_s \rangle) = \begin{cases} \langle h_i, h_s \rangle, & \text{if } a_e > 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \\ \left\langle \lfloor a_v + \frac{1}{2} \rfloor, 0 \right\rangle, & \text{if } \frac{1}{4} < a_e \leq \frac{1}{2} \wedge \langle h_i, h_s \rangle = 0 \\ \langle g_i, 4g_s + 1 \rangle, & \text{if } a_e \leq 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \wedge \\ & \max(\langle h_i, h_s \rangle) - a_v 2^{\exp(\langle h_i, h_s \rangle)} > \frac{1}{2} \\ \langle g_i, 4g_s + 3 \rangle, & \text{if } a_e \leq 2^{-(\exp(\langle h_i, h_s \rangle)+1)} \wedge \\ & \max(\langle h_i, h_s \rangle) - a_v 2^{\exp(\langle h_i, h_s \rangle)} < -\frac{1}{2} \\ \langle g_i, 4g_s + 2 \rangle, & \text{otherwise}, \end{cases}$$

where

$$\begin{aligned}
\langle g_i, g_s \rangle &= H(\langle a_v, 2a_e \rangle, \langle h_i, h_s \rangle) \\
\max(\langle h_i, h_s \rangle) &= \begin{cases} h_i, & \text{if } h_s = 0 \\ 2\max(\left\langle h_i, \lfloor \frac{h_s}{4} \rfloor \right\rangle) - 1, & \text{if } h_s \equiv 1 (\mathrm{mod} 4) \\ 2\max(\left\langle h_i, \lfloor \frac{h_s}{4} \rfloor \right\rangle), & \text{if } h_s \equiv 2 (\mathrm{mod} 4) \\ 2\max(\left\langle h_i, \lfloor \frac{h_s}{4} \rfloor \right\rangle) + 1, & \text{if } h_s \equiv 3 (\mathrm{mod} 4) \end{cases} \\
\exp(\langle h_i, h_s \rangle) &= \left\lceil \frac{|h_s|}{2} \right\rceil,
\end{aligned}$$

is poly-time, computed via bounded recursion on the notation of a_e (the bound is not explicitly specified, but h_s only grows by two bits for every bit of precision in a_e and h_i is bounded by $\lfloor a_v + 1 \rfloor$ or the previous value of h_i).

Together with the function

$$F(\langle h_i, h_s \rangle) = \Phi^{\ddagger}(b\lceil m, 2^{m+1})$$

for

$$\begin{aligned}
\Phi^{\ddagger}(B,n) &:= \begin{cases} \langle 0,\infty\rangle, & \text{if } n=0 \\ \textbf{try } \left\langle \Phi^{\dagger}(B,\lfloor \frac{n}{2}\rfloor), 2^{-\left|\lfloor \frac{n}{2}\rfloor\right|}\right\rangle \\ \quad \textbf{catch}(x)\ \Phi^{\ddagger}(B,\lfloor \frac{n}{2}\rfloor) \end{cases}, \quad \text{otherwise} \\
m &:= \exp(\langle h_i, h_s\rangle) \\
b\lceil m &:= \lambda n. \begin{cases} b(n), & \text{if } n\le m \\ \textbf{raise } x, & \text{otherwise} \end{cases} \\
b(n) &:= \mathrm{dya}(\mathrm{man}(g(n)), \exp(g(n))) \\
g(n) &:= \left\langle h_i, \lfloor h_s 4^{n-m}\rfloor\right\rangle .
\end{aligned}$$

(where $\Phi^{\dagger}$ is a version of Φ that honors the new exception x) they form a storage-p.a.r. of the function ϕ.

The M part of the modulus can be constructed exactly as in Theorem 7.6, and the bound N is

$$N(A,m,n) = \left\langle \lfloor A(m(0)) + 2\frac{1}{2}\rfloor, 2^{2\left|\max_{i\le |n|} A(m(2^{|n|-i}))\right|}\right\rangle .$$

(the maximum can be computed in **BFF** as shown in [13]) Because of the properties of the encoding, if $A(|n|-i)_e \le 2^{-k}$, $|A(|n|-i)| \ge 2^k$, and since in H we're adding two bits for every bit of precision in the approximation, N gives us a bound for the size of the history information. □

Using the properties of the encodings that we have requested in this chapter, Theorem 6.4 defines a poly-time conversion between storage- and recursion-p.a.r., thus the results of this section also apply to recursion-p.a.r.'s.

7.5 Function complexity on closed intervals.

The previous sections give correspondence between complexity in this model and type-2 complexity. In this section we will compare our approach to complexity on closed intervals from the domain in the sense of Ko. To do this, we introduce uniform moduli on closed subsets of the domain (which are also uniform moduli of continuity for the function and its representation):

Definition 7.6 *A* uniform modulus *on* $[a,b] \subseteq$ dom ϕ *of a p.a.r.* F *of a real function* ϕ *is a function* $U : \mathbb{N} \to \mathbb{N}$*, such that*

$$\forall \alpha \in [a,b] \forall a \in \mathbb{A}_{\alpha} \forall k \forall n (a_e \le 2^{-|U(k)|} \to (F(a))_e \le 2^{-|k|})$$

Theorem 7.8 *A partial real function* ϕ *has a sharp CF-representation on* $[a,b] \subseteq$ dom ϕ *in a hereditarily self-majorized class of type-2 functionals closed under composition and functional substitution for total functions if and only if it has a p.a.r. and a uniform modulus in the same class.*

Proof(→). Use a and b to find an upper bound for the absolute value of α, then apply the same reasoning as in Theorem 7.6. □

Proof(←). $M(A, m, k) = m(U(k))$ is a modulus for all A's representing reals in the interval, thus ϕ is p.a.r.-computable in the class and thus has a sharp CF-representation in it. □

This definition characterizes real functions by a pair of type-1 functions in a manner similar to Ko's approach in its version that applies to the first Grzegorczyk characterization of real functions (given by Definition 2.11). Let us take a look at a few instances of this theorem:

Corollary 7.1 *A partial real function ϕ has a CF-representation primitive recursive in the sense of Kleene ([50]) on $[a, b] \subseteq$ dom ϕ if and only if it has a primitive recursive p.a.r. and a primitive recursive uniform modulus on $[a, b]$.*

Corollary 7.2 *A partial real function ϕ has sharp CF-representation in **BFF** on $[a, b] \subseteq$ dom ϕ if and only if it has a poly-time p.a.r. and a poly-time uniform modulus on $[a, b]$.*

It is easy to see that the following fact is true (using Ko's characterization of the poly-time real functions as poly-time maps of rational approximations with poly-time moduli of continuity):

Theorem 7.9 *A partial real function ϕ is feasible in the sense of Ko on $[a, b] \subseteq$ dom ϕ if and only if it has a poly-time p.a.r. and a poly-time uniform modulus on $[a, b]$.*

Combined with the previous corollary it gives an alternative proof of Theorem 5.2.

Chapter 8

Efficient Implementation of Real-Number Arithmetic

This chapter starts the practical part of the thesis, where we will discuss the *RealLib* library for exact real number computations and a component of the package that can also be used as a stand-alone library for double precision interval arithmetic.

Portions of this chapter are also in [77], *"RealLib, an Efficient Implementation of Exact Real Arithmetic"*, CCA 2005 — Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan. Informatik Berichte 326-7/2005 FernUniversität Hagen, Germany 2005.

8.1 Introduction

In developing a library[1] for exact real number computations, our main objective has been to create a tool which is useful in a wide variety of contexts. For this, the library needs to be able to replace standard floating point with a minimum of extra programming, stay clear of bad programming practices to decrease the possibility of the library introducing bugs in existing code and to facilitate the understanding of the mechanisms of the library, and, maybe most importantly, the library must be able to reach performance comparable to that of hardware floating point in the cases where it is actually possible to compute meaningful results using low precision.

Combining these requirements presents a very difficult task. The performance requirement clearly excludes higher-level approaches that manipulate the real numbers as entities (more information about the performance problems of this approach will be given below; examples of packages implementing this include *ICReals* [20], *XRC* [7], *Few Digits* [12] and others). On the other hand, a minimal user effort in replacing floating point arithmetic with real number arithmetic appears to require such an approach. It may seem that a user interface that pretends to act on complete objects while in reality it does something else is an answer to the problem, but such an approach appears to require bad programming and non-standard behavior of the user's programs as

[1] *RealLib*, available at `http://www.brics.dk/~barnie/RealLib/`

demonstrated by Norbert Müller's *iRRAM* [86].

Our exact real number implementation takes an approach which provides two levels of access to real numbers aiming to satisfy conflicting parts of the requirements. One of the levels operates on real numbers as complete entities and is able to be easily integrated in existing code, but has poor performance when a multitude of simple operations is to be performed. For the latter case, our design provides an interface which operates on the level of approximations, is free of performance issues, but where the control flow may become more complicated as the objects operated on do not represent complete reals. Both levels have well defined behavior and do not use any non-standard programming.

The two levels interact with each other, more specifically, the layer that operates on complete entities (to be called the "top", "numbers" or "direct" layer in the rest of the chapter) can use objects written in the layer that operates on approximations (to be called the "bottom", "function" or "approximations" layer). With the help of this mechanism, a user may encapsulate large computations in a function on the bottom layer, and use it once or repeatedly at the direct layer. In an extreme case (which may happen quite often in practice and is very similar to the mode of operation of *iRRAM*), all of a computation may be implemented on the approximations layer as what we call a "nullary" function (i.e. a real number constant), using the top layer, for example, to only create a real number linking to that object and print out an approximation to it.

The rest of this chapter will explain the obstacles in designing efficient real number systems along with suggestions for solving them, explain the way these solutions are used in *RealLib*, and compare the library to the best available alternatives.

8.2 Performance problems in real number arithmetic

The approach to real number computability most often attempted in practice is some kind of type-2 approach using the TTE model of Weihrauch [113] or an equivalent formulation. The approach looks pretty easy to implement, especially if one uses a functional language such as *SML*, *ocaml* and others.

Unfortunately, the type-2 character of the approach presents some barriers to efficiently implementing the ideas. Some of them can be avoided, but some of the problems are so serious that a clean type-2 implementation free of very serious performance problems is impossible to achieve.

Indeed, clean[2] type-2 implementations have proven to be extremely slow, being at least a magnitude slower[3] than mixed solutions such as *RealLib* and *iRRAM*[4], even in cases where higher precisions are used and the bookkeeping overheads are low. What are the reasons for this poor performance?

[2]The term will be properly defined below.

[3]This statement is based on the results of Section 8.5 and the "Many Digits" friendly competition, `http://www.cs.ru.nl/~milad/manydigits/`, in comparing the exact real number packages *RealLib* and *iRRAM* to *XRC*, *Few Digits* and *BigNum*.

[4]There is a common misconception that *iRRAM* is an implementation of the TTE model of real number computability. This is not true, as the reader will see later in Section 8.6.

Since every real number used in the course of a computation needs to be present as a function object, a type-2 approach requires the library to use representations of the way in which a real number was obtained, which has to be implemented in some structure that stores a term representation of every object used in the computation.

This leads to a wide variety of performance problems which will be discussed below.

8.2.1 Problems with common subexpressions

The type-2 implementation of a binary operation computes an approximation based on approximations to its operands at higher precision. Every time a binary operation is called, it asks its first argument to compute an approximation, then it asks its second argument to compute an approximation and uses them to compute an approximation to the output.

If both arguments are the same object, a straightforward implementation would require that the same computation (probably with different accuracy) be carried out twice. Moreover, a naive implementation would build a representation of the computation as a tree and will contain the argument twice.

Consider this simple example using a hypothetical class *Real* that implements type-2 real number arithmetic:

```
Real a(1);
for (i=0;i<100;++i)
    a = a+a;
```

If the class *Real* uses a tree to store the term representations of the real numbers, at the end of this fragment *a* will refer to a tree with $2^{101} - 1$ nodes.

A better implementation of the class should prefer to use a directed acyclic graph (dag) in the representation of real numbers to allow multiple references to the same node and avoid the unnecessary growth of the size of the structure.

Even after switching to dags to represent expression trees, the better implementation may run into the problem of complexity explosion, because the straightforward implementation would still require the computation of an approximation to each node twice for every addition, or the computation of 2^{100} approximations to 1 and $2^{100} - 1$ additions on approximations. Naturally, such an explosion cannot be permitted.

The problem can be solved by careful caching of the approximations, making sure that the approximations to a node are asked with the same precision, and that the cached approximations are deleted when they are no longer needed. This is highly non-trivial to do in a clean type-2 implementation.

8.2.2 Unnecessary precision growth

A type-2 implementation of a function requires higher accuracy from its arguments in order to be sure about the accuracy of its result. This may lead to unnecessary big precision growth.

Consider this code fragment:

```
Real a(0);
for (int i=1;i<=1000000;++i)
    a = a+Real(i);
```

Addition requires approximations to its arguments of accuracy at least one bit higher. Thus in this code fragment the accuracy needed will grow by at least one bit in every iteration. Thus, if we want to compute an approximation to a which is 32-bit accurate, we will end up performing many of the additions at very high precision, some of them with 1000000 or more bits of precision. Clearly this is not acceptable from a performance point of view, since in reality less than 60 bits suffice to carry out the whole computation.

One may wonder if it is possible to balance the precision request so that a binary operation requires less precision from its more difficult arguments. We do not see how this could be implemented in practice.

The approach described so far is sometimes called "top-down evaluation" to indicate that a node controls the accuracy of its siblings. The name also hints at the possibility of a "bottom-up" approach where the accuracy of the siblings determines the accuracy in a node, an approach which does not suffer from the problem at hand.

The bottom-up approach evaluates everything at an (almost) constant precision starting from the leaves of the tree, keeping track of the errors that are introduced at every step. In certain cases it may turn out that the end result of the computation on a tree is not accurate enough. In such a case the whole computation is restarted at higher precision until the process leads to a result which is accurate enough (i.e. in which the accumulated error is sufficiently small). The reiterations do not add much to the complexity of the process, since e.g. by doubling the precision for each new iteration we can be certain that time taken by the evaluation is dominated by the last iteration.

In this case the implementations of functions use interval arithmetic and approximations to transcendental functions that accept arbitrary precision requests. It may seem that such an approach requires at least twice the amount of work, when one considers that an interval is represented as a pair of bounds and every function must be evaluated at least twice. This is not required, as one can use what is sometimes called "simplified interval arithmetic", where the functions are evaluated only at the center and the size of the resulting interval is estimated from the size of the input interval.

It becomes very unclear whether we can still call this a type-2 approach after this modification. We will leave this question open, and will keep using the term "clean type-2 implementation" to mean a type-2 implementation that uses the normal top-down evaluation, and will reserve "type-2" as an indication of an approach where every real number is available as a function representation and a real number function has full access to that representation.

8.2.3 Bookkeeping necessary

Even with the modifications discussed above, a type-2 implementation requires a new object to be created every time an operation is performed. If we assume

that this object takes at least 4 bytes of memory (to store the 32-bit address of an argument), the absolute maximal number of operations that can be carried out in a computation on a 32-bit machine is about 1 billion. Since this number of operations can be easily reached e.g. in linear algebra, this restriction is clearly unacceptable.

Moreover, every operation must allocate memory, which is known to be a painfully slow process in modern computers, significantly slower than the time it takes to actually perform the operation at low precisions. If we want to be able to reach the performance level of hardware floating point, this is clearly unacceptable.

Therefore one must consider an approach that does not store the history of a computation at every step. This is not possible in a type-2 approach, since the basic property of real functions in a type-2 model is the access to full function representations of their real arguments.

The functions must operate on approximations and be modular in the approximations in the sense that the representation of the composition of two functions must be achieved by composing the representations of the two functions. In order to also achieve full soundness and completeness, i.e. equivalence to the popular notions of computable analysis, a type-1 theoretical approach must be used. Combined with the requirement for bottom-up evaluation, an interval approach such as Grzegorczyk's interval definitions [34] of real function computability must be used.

8.2.4 Loss of locality information

Even if we disregard the time and space problems of creating a term representation, which may be negligible at higher precisions, a type-2 representation of a computation lacks the locality information that a programmer or compiler gives in a implementation of a function.

Let us take a look again at a slight modification of the last example:

```
Real a(0);
for (int i=1;i<=1000000;++i)
    a = Real(i)+a;
```

Depending on the actual way that the process of evaluating the generated expression is implemented, this computation may require the storage of up to 999999 temporary values. The evaluation of a tree is recursive, and a naive implementation may evaluate the left hand side argument to additions first before diving into the recursion to compute the right hand side argument, which will lead to computing and storing a value for every iteration. If the precision is high, this process will waste huge amounts of memory. Since this will also destroy all data locality and trash all cache levels, the performance in such a case is very far from acceptable.

While it is easy to find a solution to the problem for this concrete example, finding an approach which chooses the better pattern of evaluating arguments does not seem trivial. A heuristic must be used which will most probably fail in a large number of cases.

To solve the problem completely, we would prefer to execute the operations with the order and locality that the programmer and compiler give. For this example, it would mean that only a single object, *a*, needs to be stored to complete the evaluation.

To achieve this, one again must use a bottom-up evaluation scheme combined with a modularity requirement on the level of approximations, so that the loop above can start with an approximation to *a*, update it at every iteration, and finish with an approximation to the end result. If that end result is not accurate enough, we should be able to rerun the code to achieve a better approximation.

8.2.5 Impossible compiler optimizations

Whenever a computational tree determines the order of computations, it is impossible to perform any compiler optimizations on the code.

In the case of very low precisions the overhead of a function call and inability to execute computations in parallel may result in two-digit factors of slowdown. We certainly do not want this if we want to be able to achieve performance close to hardware floating point when the problem to be solved is easy.

To achieve the best possible performance, the programmer must be able to write function code that is compiled specifically for fast instantiations of interval arithmetic and additionally for slow but generic multiple precision floating point. In *C++* this is achieved using template functions.

The library must permit this tool to be used to achieve optimal performance.

8.3 Design of the **RealLib** library

The *RealLib* library provides two interfaces to the user. One of them behaves like a type-2 implementation of exact real arithmetic, while the other operates on approximations of numbers but still implements exact real arithmetic by being an implementation of the model of Partial Approximation Representations for computable analysis [75]. The top-level interface uses a bottom-up approach to evaluating real numbers and is thus not a clean type-2 implementation. It does this to avoid the first two performance problems discussed in the previous section.

Our bottom-up approach uses fixed precision for all the computations on an expression dag, which not only lets us avoid the precision growth associated with top-down evaluation, but also makes it easier to avoid the complexity explosion by ensuring that once an approximation to an object is computed, this approximation will have the exact precision needed for all other references to the same object. By counting their number and the number of requests already made, we can maintain efficient caching of all temporary results and delete them exactly when they are no longer needed. Additionally, in an attempt to minimize the effect of the loss of locality information, the library also tries to optimize the order of evaluation of the siblings of nodes with multiple arguments.

The top layer of the library behaves like a built-in type for floating point arithmetic with the exception of the aspects that have been proven to be undecidable: because of the non-computability of the equality test, all comparisons of real numbers are undefined for equal arguments; in consequence to this, the library does not provide non-strict versions of comparisons (<= and >=) because they coincide with their strict counterparts; additionally, the rounding in the library is always faithful (up to a distance of 1 unit in the last place) as correct rounding (such as rounding to a double precision number according to the IEEE-754 rules) is not achievable.

The top layer of the library is free of the first two performance issues, but suffers badly from the rest of the problems discussed in the previous section. In particular, there is a need to maintain full information about the way in which a number was constructed. This is very inefficient when simple operations are involved.

To solve this problem, the library provides a bottom layer, the level of approximations where the objects on which the user's code operates are interval approximations. The objective of the bottom layer is to represent big chunks of a computation tree as single nodes by providing a function that encapsulates a lower-level version of the code of the same operation. This is made possible by the bottom-up approach for evaluation and the theoretical model, which also gives us representations of all computable real functions on the level of approximations in a modular way. The latter allows the approximations layer to look as if it is working on complete real numbers.

The bottom layer does not store any additional information about the temporary real numbers other than their approximations and performs computation exactly in the order and locality given by the programmer and compiler. Moreover, the bottom layer functions are always defined as template functions, so that a very fast double precision step can be used for the first approximation.

This "function" layer is used to define functions that can be used directly on the numbers layer. For example, it is easy to define a new function computing the Riemann ζ of a complex number, and use this function repeatedly with both the top and bottom layer interface.

An actual computation starts when an object is created on the top layer and a request for a property of it is given (such as e.g. a 10-digit decimal approximation). The request triggers a recursive evaluation on the description of the number at a chosen precision, which in turn executes the bottom layer functions used in the definition. They may be executed more than once, since an end result may turn out to be insufficient to show the property, or the functions may be abruptly terminated by an exception requesting higher precision from a function used in their body (such as division when the current approximation of the divisor does not separate it from zero).

With the combination of the two layers we have a mixed implementation in which the results can be extracted via the more convenient type-2 interfaces, while the bulk of the computations can be carried on the much more efficient approximations level. We still have full descriptions of all temporary objects used in the numbers level, but they do not need to be as many since a single node can encompass a multitude of simple operations. In this case, the program

code written on the function layer of the system becomes part of the description of the term used to obtain a real number.

In clean type-2 implementations of real arithmetic, the single nodes can only be representations of the functions that are built into the system. Even if the system allows it, adding a new function to the set is not a trivial task, since the approach requires a careful control over the accuracy which is not automatically available. With *RealLib* in many cases adding a function to the set of objects working on the fast layer of approximations is achieved by simply changing a function's header and using a linking macro (as the examples in Section 8.5 show).

A type-2 interface is the most convenient method of working with real numbers, but unfortunately it is not very efficient unless a huge library of predefined functions is available. While we are not able to provide that library for *RealLib*, we have made it as easy as possible for the user to add new functions that run as close to the hardware as possible.

8.4 Limit computation and approximate comparisons

The results of applications of most interesting functions in analysis are given numerically as limits of computably converging sequences of computable numbers. One of the ways to define a computably converging sequence is by giving a sequence together with an estimation of the amount of error in all the approximations of the sequence. This corresponds to giving a partial approximation representation (see [75]) of the limit.

The method used in *RealLib* to define limits follows this idea. A function written on the level of approximation can add to the amount of uncertainty in an approximation to cover for uncalculated portions of the result. Every such function is given a parameter *prec* that specifies a precision; if a function needs to compute the limit of a sequence it needs to

- generate members of the sequence for different values of *prec*,
- indicate the distance within which the limit must be contained for every member of the sequence,
- make sure that for every target accuracy ε as *prec* grows there will be a point after which the distance is always smaller than ε.

For example, if one tries to compute the number e by evaluating its Taylor series expansion, one can choose to evaluate the first *prec* number of elements and find a bound for the remainder sum which is to be added to the error in the approximation of the result. Since as *prec* grows this function gives improved approximations to the number, the third condition is also satisfied.

The same method is used in the implementations of real number functions that need to compute a limit. The only difference is that *prec* is no longer explicitly given as an argument to the function. In exact correspondence with the theoretical model, this parameter is recovered from the approximation to one of the real number arguments of the function.

Examples of defining a function that requires limitation are given in the library's manual [78].

Another operation which is a requirement for the completeness of a real number package, is the presence of approximate comparisons, i.e. a method of showing either $x < b$ or $a < x$ for an arbitrary x when $a < b$. This is not directly given in our library, but the approximations level of *RealLib* includes comparisons that only evaluate to true if the current approximation is sufficiently accurate to prove the fact. Using a combination of two such comparisons and forced reiteration if neither of them is true, one can easily achieve the approximate comparison operation.

In addition to this, the library provides "weak" operations which can be used to choose more efficient execution paths. A weak operation only evaluates properties of the center of an approximation and is not guaranteed to give consistent results in consecutive iterations through the same code. A *weak_round* operation, for example, can be used to reduce an argument to a periodic function to its primary domain. Although the same real number may round to different values in different iterations, this does not lead to problems as all possibilities will ultimately lead to the same result as a real number.

8.5 Examples and performance comparison

We will give a few sample programs written for the library, starting with two cases where it is known a-priorily that the computations cannot be correctly handled in machine precision, and finishing with a case which shows the strength of the library in dealing with the situations that more often appear in practice: where the need for high-precision computations may be suspected, but hardware floating point actually suffices.

The first example is a very simple demonstration of a feature all exact real number systems share: the possibility to request arbitrarily precise approximations to a number. In this case, we choose to display a 10000-digit approximation to the value of π:

```
001 #include <iostream>
002 #include <iomanip>
003 #include "Real.h"
004 using namespace std;
005 using namespace RealLib;
006
007 int main(int argc, char **argv) {
008     InitializeRealLib();
009     {
010         cout << setprecision(10000) << Pi;
011     }
012     FinalizeRealLib();
013     return 0;
014 }
```

The actual work of this code is done at Line 10, the rest of the file includes the appropriate headers, makes the definitions in the *std* and *RealLib* namespaces local, and takes care of the necessary initialization and finalization

of the library. At Line 10, *Pi* is a predefined value for the library and represents the exact value of the real number π via a function that computes approximations to it for any given precision. To display the result with the number of digits specified by *setprecision*, the library will call this function, possibly more than once, to get an approximation accurate enough to display 10000 digits which are no further than a unit in the last place from the actual value.

We will not print the result here, instead we will measure the time[5] it takes to complete this program and compare it to two other exact real number systems, *XRC* by Keith Briggs [7] and *iRRAM* by Norbert Müller [86]:

*RealLib*3	*iRRAM* 2004_02	*XRC* 1.1
730 ms	230 ms	364 s

iRRAM has had the reputation of the fastest exact real number library, using highly optimized *GMP* [116] for the higher precisions that are required for this example. It does not fail it in this case, producing the approximation three times faster than *RealLib*, which still uses a portable custom multi-precision library written entirely in *C++*.

XRC, on the other hand, is too slow.

For the next example, we will use the logistic sequence example from [86]. We will compute the iteration $x_{i+1} = 3.75(1 - x_i)x_i$ with $x_0 = 0.5$ and print 6 digits of x_{100}, x_{1000} and x_{10000}. This time, we will use two different versions of the program. One that uses only *RealLib*'s mechanism for dealing with real numbers as entities, and one that uses *RealLib*'s mechanism for constructing functions that operate on the efficient approximations layer.

We will encapsulate the computation in the following function:

```
007 template <class TYPE>
008 TYPE Logistic(unsigned int prec, UserInt len)
009 {
010     TYPE s(0.5);
011     TYPE coeff(3.75);
012     TYPE one(1.0);
013     for (int i=1;i<=len;++i)
014         s = coeff * (one - s) * s;
015     return s;
016 }
017 CreateIntRealFunction(Logistic)
```

This is a template function that has an unused argument *prec*. This form, along with the declaration at Line 17, is required by the library for functions that work on the approximations layer of the library. This example does not use the argument *prec*, because we are not computing a limit of a sequence. Other than that, it's a pretty standard code for the iteration, and, being a template, it can also be used to try the direct implementation of the computation using the type *Real*, which is the basic type in the library for working with real numbers as entities. We will make use of this in the following main function (not much different from the one in the previous example):

```
019 int main(int argc, char **argv) {
020     InitializeRealLib();
021     {
```

[5] using a Pentium-M 1.8GHz and GCC 3.3 in Cygwin environment

```
022         cout << Logistic(1000) << endl;
023     }
024     FinalizeRealLib();
025     return 0;
026 }
```

Line 22 is the important one, which in this case calls the real number function object constructed at Line 17 from the template function *Logistic*. This object has a single argument, because it makes no sense to specify precisions for exact computations. In the table that follows this will be reflected in the column "*RealLib*3, function". For the column "*RealLib*3, direct" we will use the same code with Line 22 changed to

```
022         cout << Logistic<Real>(0, 1000) << endl;
```

which runs an instantiation of the template function directly using the type *Real*, and for the column "double" we will use

```
022         cout << Logistic<double>(0, 1000) << endl;
```

Timing these[6] and corresponding code for the other two exact real number systems results in the following table:

iterations	*RealLib*3, function	*RealLib*3, direct	*iRRAM* 2004_02	*XRC* 1.1	*double*
100	1 ms	3 ms	625 ms	383 ms	0.6 μs
1000	150 ms	188 ms	12 ms	143 s	6 μs
10000	48 s	50 s	5.5 s	–	60 μs

For unknown reasons *iRRAM* did not want to compute the 100 iterations as fast as we would expect, thus we suggest that the reader ignore the first value in *iRRAM*'s column[7]. *XRC* apparently was not expected to be used for heavily nested computations and its recursive evaluation mechanism failed for more than several thousand nested operations.

iRRAM is again the fastest, and *XRC* is disappointingly slow. All libraries compute the correct values in contrast to the double precision implementation, which runs really fast, but is completely wrong.

What is interesting in this example, is that the overhead of using the top-level interface of the library in comparison to the approximations interface, is clearly visible. It may be little or negligible at high precision (only 2 out of the 50 seconds required to compute the 10000 iterations), but it is quite a big portion of the time for a lower precision computation, and dominates the time in the simplest case, taking twice as much time as the actual computation!

Because of the problems inherent in a type-2 approach to exact real arithmetic, we do not believe that there is a chance to improve the type-2 overheads much further than what has already been done in *RealLib*. Instead, seeing results similar to the ones above, we opted for incorporating user functions that use an interval approach as an option that could combine the ease-of-use of higher-type access to the numbers with the efficiency of lower-type user functions. This decision was also influenced by the fact that *iRRAM* already employed an approach that works on approximations and was displaying its strengths.

[6] To measure execution times in the order of microseconds, the timing is done using a modification of this code that executes Line 22 many times.

[7] The author of *iRRAM* could not supply an explanation or remedy for the problem.

For the next example, we will do a computation that does not require high precision: the first 6 digits of the sum of the first 1000, 10000, 100000 and 1000000 members of the harmonic series $\sum_{i=1}^{n} \frac{1}{i}$. These values can be correctly computed in double precision. We will use the following function:

```
007 template <class TYPE>
008 TYPE Harmonic(unsigned int prec, UserInt len)
009 {
010     TYPE s(0.0);
011     TYPE one(1.0);
012     for (int i=1;i<=len;++i)
013         s += one / i;
014     return s;
015 }
016 CreateIntRealFunction(Harmonic)
```

Similarly to the previous example, we measure the time needed when a function object is used and the time needed when the function is directly instantiated to the types *Real* and *double*, as well as corresponding code for the other two libraries. Let us see how exact real number packages compare to hardware floating point:

members	*RealLib*3, function	*RealLib*3, direct	*iRRAM* 2004_02	*XRC* 1.1	*double*
1000	91 μs	27 ms	1.3 ms	22 ms	21 μs
10000	580 μs	275 ms	12.5 ms	–	212 μs
100000	5.5 ms	2.85 s	118 ms	–	2.23 ms
1000000	54 ms	28 s	1.2 s	–	22 ms

Again, *XRC* failed to produce result with 10000 or more nested operations. *iRRAM* was about 60 times slower than hardware floating point.

The two layers of *RealLib* show radically different results. While the high level approach has performance as disappointing as a factor of 1000 slower, the implementation of the computation as a function using the library's approximation interfaces achieves a performance which is not that different from the hardware double precision and significantly faster than any other previous implementation. Moreover, the implementations of the same function on the two levels only differ in the types used in the definition of the function (*Real* vs. a template argument), an unused argument in the header of the function, and the use of a declaration to link the two layers.

If we switch to a better compiler[8][9], the distance between exact real numbers using *RealLib* and hardware floating point disappears completely:

members	*RealLib*3, function	*RealLib*3, direct	*double*
1000	22.5 μs	3 ms	18.5 μs
10000	186 μs	30 ms	185 μs
100000	1.75 ms	1.05 s	1.85 ms
1000000	17.8 ms	5 s	18.5 ms

Although the current version of *RealLib* is slower than *iRRAM* when higher

[8] Pentium-M 1.8, Intel C++ Compiler 8.0, Windows XP

[9] unfortunately, *GMP*, *XRC* and *iRRAM* do not support Windows natively

precisions are needed, running in a general context it will achieve significantly better performance on average, because the majority of the problems that show up in practice are easy, and the performance of the library in the error-sensitive cases is of the order of magnitude of the fastest alternative, much closer to it that any other implementation.

8.6 Relation with iRRAM

Although it is commonly viewed as an implementation of the TTE model for computable analysis, which is a typical type-2 model, one can easily see that the *iRRAM* cannot be called a type-2 implementation in any sense used in this paper. We do not believe it can be called type-2 in any sense whatsoever, since the only connection is using a theoretical foundation which is *equivalent* to TTE.

The author of the *iRRAM* defines the library to be a *simulation* of Brattka and Hertling's feasible Real RAM [6], an algebraic definition of the computable real numbers equivalent to the TTE. Because the programs of the *iRRAM* actually operate on simplified interval approximations and must permit reiterations, the library is a simulation rather than an implementation of the model.

Programs written for the *iRRAM* operate only on the level of approximations, using a bottom-up evaluation scheme and modularity on the level of approximations. As such, they only simulate operations on complete objects, and at no point in time do they have access to the functions that define real numbers. Thus *iRRAM* is, indeed, a type-1 implementation of exact real arithmetic.

Because of this main characteristic of the *iRRAM*, it does not suffer from the performance problems discussed in this paper except the inability to use compiler optimizations to implement very fast initial approximations.

There are two ways of using *iRRAM*. Either the user's program is completely under the control of the system, or the program uses *iRRAM* occasionally for separate computations.

The former can be very difficult to control, because the execution of a user's program under *iRRAM* becomes complicated by possible reiterations and abrupt termination. In this mode, all user code is executed a multitude of times, not only the portions that use exact real computations.

The latter mode of using the system is far more inconvenient than having a type-2 interface to the real numbers. We may be forced to redo the same computation more than once if, e.g. we want to print a variable as an absolute value and its base-10 logarithm.

RealLib takes the ideas of the *iRRAM* and pushes them further. Here is a comparison of some of the key features of the two libraries:

- Both operate on interval approximations, but the version of *iRRAM* which was available at the time of this performance comparison did not include a fast initial step using hardware floating point. As communicated by the author, at least the development versions of the *iRRAM* now include such a step, but it is realized using run-time choices to switch between

hardware interval arithmetic and software multi-precision simplified arithmetic. *RealLib* makes that a compile-time choice which allows compiler optimizations and performance very close to hardware floating point. This also makes it easier for *RealLib* to include additional optimized step in the future, such as a double-double or quad-double evaluation [37] as a second iteration, or possibly a fast implementation using integer arithmetic with hard-coded precision.

- When incorporated into bigger programs, computations using the *iRRAM* do not integrate well; *iRRAM* has to use tricks such as overriding the standard *C++* input/output streams in order to pretend to operate on real numbers as complete objects, while in fact the methods used only have access to approximations. *RealLib* provides a type-2 interface to allow exact computations to be used in a program without modifying its behavior and control flow, while the efficient layer that corresponds to the mechanisms used in *iRRAM* is clearly marked as a layer that operates on approximations. Still, in the cases where a function is a sequence of already defined operations without conditionals based on the values of real arguments, the program code on the approximations layer looks and behaves as if it works on complete entities, because the functions and operations applied on this layer are in fact implementations of the corresponding real number functions.

- *iRRAM* asks for a CF-representation of a computably converging sequence to define its limit, while *RealLib* relies on accounting for the error of the approximation. This is defined by the theoretical model used in the two systems, but we feel that the latter gives us a more uniform approach, since the former is in effect a case of top-down evaluation of the limit, something both libraries try to avoid.

- The template approach of *RealLib* and the availability of computation dags allows the library to accommodate separation bounds [84], which in a future version of the library could allow to decide equality for algebraic numbers.

- The multiple-precision back-end of *iRRAM* is *GMP*, which is very fast and hand optimized using assembler code. *RealLib* uses a custom library which is completely portable, but is much slower. The only thing that prevents *RealLib* to use the lowest level functions of the faster *GMP* as back-end is the fact that we give higher priority to the computations at low precision which appear much more often in practice and we have not yet implemented the link between the library and *GMP*.

- Intensional (also known as multi-valued) functions are an integral part of *iRRAM* and currently not supported in *RealLib*. They require a theoretical background which was only recently developed (see [75]) and has not been realized yet.

- Unlike other packages that do not allow the computation of limits in user programs, both *RealLib* and *iRRAM* can be shown to be complete, i.e. able to define all computable real numbers and extensional functions, by showing that all operations in the feasible Real RAM model are defined.

Chapter 9

Interval arithmetic with Intel's SSE2

9.1 Introduction

Interval arithmetic is a very useful tool that can be used to partially solve the problem of roundoff errors or as part of a complete solution in the form of exact real arithmetic.

RealLib relies on fast machine precision interval arithmetic for its first stage. The performance of the library in the cases that most frequently appear in practice, where machine precision suffices, depends only on the performance of the first stage. Thus it is crucial to have a very fast implementation of interval arithmetic.

The IEEE-754 standard for floating point arithmetic [117] has useful features to aid fast interval arithmetic, namely the directed rounding modes that should be present with every IEEE-754 implementation. Unfortunately, in some processor architectures, notably Intel's x86, it is non-trivial to effectively use them, as switching the rounding mode for an operation requires significantly more time than the operation itself. Even when one takes into account the fact that one of the directed rounding modes can be emulated by operations on negated values rounded in the other direction, an interval arithmetic package has to be aware that users may mix interval with standard floating point arithmetic and would still require repeatedly switching the rounding modes.

Fortunately, the newer generations of the x86 architecture provide an additional set of registers with its own rounding control, the SSE2 double-precision floating point registers [118]. They can coexist with the old x87-style floating point, which is still the register and instruction set used most widely. Thus, to serve all purposes, we can reserve the SSE2 register and operation set for interval arithmetic and leave x87-style floating point for any standard floating point operations that the user may be performing.

The SSE2 instruction set can also work on packed data, as every SSE2 register can contain and operate on a pair of double-precision floating point numbers. Since an interval is in fact a pair of bounds, one SSE2 register can be used to hold an interval, which nullifies the additional register pressure that interval arithmetic would normally exert.

With this it is possible to develop a very fast machine precision interval

arithmetic implementation. *RealLib* uses such an implementation which will be detailed in this chapter of the thesis.

As it is part of an exact real arithmetic package, the objective of this implementation is more oriented towards performance rather than accuracy, i.e. it prefers overestimating an interval rather than investing too much time in evaluating it tightly. We believe that this time would be better spent at the next iteration at higher precision, which would happen only if the computation actually requires it.

Additionally, the implementation ignores the portions of the argument of an operation that are outside its domain, e.g. the negative parts of the argument in a square root, meaning for example that $\sqrt{[-1,4]} = [0,2]$. This is the proper mode of operation to ensure that $\sqrt{0}$ is computable in exact real arithmetic.

9.2 Key ideas

Normally, interval arithmetic based on floating point would use two rounding operations, Δ (rounding towards $+\infty$) and ∇ (rounding towards $-\infty$). By default IEEE-floating point uses rounding to nearest, which is not useful for our purposes.

We already mentioned that switching the rounding mode has a detrimental effect on the performance of floating point operations, thus we would want to avoid all rounding mode switches. We will only do this once, at the beginning of a computation[1], setting the rounding mode to rounding towards $-\infty$. To compute lower bounds of the results, we will directly use the floating point operation. To compute upper bounds, we will make sure that the result of the floating point operation is negated, thus making use of the identity

$$\Delta(x) = -\nabla(-x).$$

Seeing operations in the form above, compilers are usually overzealous[2] to fold the pair of negations and destroy the effect we want to achieve. To avoid this, at the same time keeping down the number of required operations, we make sure that we always keep the high bound of the interval negated, i.e. our representation of the interval $x = [\underline{x}, \overline{x}]$ is the pair $\langle \underline{x}, -\overline{x} \rangle$. (in the rest of this chapter we will assume every interval is represented in this fashion and will simply write x to mean $[\underline{x}, \overline{x}]$ and $\langle \underline{x}, -\overline{x} \rangle$)

Three observations can be made directly from this:

- in this setting, the sum of x and y is evaluated by $\langle \nabla(\underline{x} + \underline{y}), -\nabla(-\overline{x} - \overline{y}) \rangle$ which is achieved by a single instruction, *_mm_add_pd*.
- changing the sign of an interval x is achieved by simply swapping the two bounds, i.e. $\langle -\overline{x}, \underline{x} \rangle$, achieved by a single instruction, *_mm_shuffle_pd*,

[1] This is accomplished by the construction of a special object that also takes care of restoring the previous rounding mode after the interval computation has completed.

[2] The two negations have no effect on the rounding-to-nearest mode which is normally in place in *C/C++* code, and on which many standard functions rely, thus this optimization is perfectly legal. Only our specific (non-standard) use of floating point makes it unwanted.

- joining two intervals (i.e. finding an interval containing all numbers in both, or finding the minimum of the lower bounds and the maximum of the higher bounds) is performed as $\langle \min(\underline{x}, \underline{y}), -\min((-\overline{x}), (-\overline{y})) \rangle$ in a single instruction, *_mm_min_pd*.

The latter is used extensively in the computation of multiplication, division and other operations.

9.3 Operations

In this section we will give short remarks on our implementation of the basic operations on intervals. The operations include the arithmetic operators, including the special cases $-x$, $\frac{1}{x}$, and x^2, absolute value and square root.

All the operations give tight bounds (i.e. the best possible enclosures after rounding).

9.3.1 Addition

Definition:

$$x + y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \subseteq \langle \nabla(\underline{x} + \underline{y}), -\nabla((-\overline{x}) + (-\overline{y})) \rangle$$

Addition is implemented as a single *_mm_add_pd* instruction. The negated sign of the higher bound ensures the proper direction of the rounding.

9.3.2 Sign change

Definition:

$$-x = [-\overline{x}, -\underline{x}] = \langle -\overline{x}, \underline{x} \rangle$$

This is a single swap of the two values, implemented as a *_mm_shuffle_pd* instruction. No rounding is performed here.

9.3.3 Substraction

Definition:

$$x - y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \subseteq \langle \nabla(\underline{x} + (-\overline{y})), -\nabla((-\overline{x}) + \underline{y}) \rangle$$

Substraction is implemented as $x+(-y)$, which corresponds to two processor instructions. This is the best that can be achieved with packed SSE2 instructions, because the formula requires a combination of the high bound of one of the arguments with the low bound of the other.

9.3.4 Multiplication

Definition:

$$xy = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})] \tag{9.1}$$

Unfortunately, the rounding steps are inseparable parts of the operations, this the equation above requires 8 multiplications. Using the fact that $\Delta(\nabla(r)+$

$\epsilon) \geq \Delta(r)$ (for ϵ being the smallest representable number), one can do with 4 multiplications at the expense of some accuracy.

In our implementation we chose a different approach where we use four multiplications without sacrificing accuracy, by selecting the multiples based on the signs of $\underline{x}$ and $\overline{x}$. More specifically, we use these observations:

$$xy = \begin{cases} [\min(\underline{x}\underline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\overline{y})], & \text{if} \quad 0 \leq \underline{x} \leq \overline{x} \\ [\min(\underline{x}\overline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\underline{y})], & \text{if} \quad \underline{x} < 0 \leq \overline{x} \\ [\min(\underline{x}\overline{y}, \overline{x}\overline{y}), \max(\overline{x}\underline{y}, \underline{x}\underline{y})], & \text{if} \quad \underline{x} \leq \overline{x} < 0 \end{cases} \tag{9.2}$$

to conclude that the formula

$$xy \subseteq \langle \min(\nabla(a\underline{x}), \nabla(b(-\overline{x}))), -\min(\nabla(c(-\overline{x})), \nabla(d\underline{x})) \rangle ,$$

where

$$\begin{aligned} a &= \begin{cases} \underline{y} & \text{if } 0 \leq \underline{x} \\ -(-\overline{y}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} -\underline{y} & \text{if } (-\overline{x}) \leq 0 \\ (-\overline{y}) & \text{otherwise} \end{cases} \\ c &= \begin{cases} -(-\overline{y}) & \text{if } (-\overline{x}) \leq 0 \\ \underline{y} & \text{otherwise} \end{cases} \\ d &= \begin{cases} (-\overline{y}) & \text{if } 0 \leq \underline{x} \\ -\underline{y} & \text{otherwise} \end{cases} \end{aligned}$$

computes the rounded results of the multiplication formula in (9.1). It uses more instructions than the direct implementation with 8 multiplications, but achieves better performance.

9.3.5 Multiplication by a positive number

When one of the numbers is known to be positive (e.g. a known constant), one can use one of the cases in (9.2) directly:

$$xy \overset{x \geq 0}{=} [\min(\underline{x}\underline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\overline{y})]$$

This is significantly faster than the general case multiplication, involving only 5 instructions (4 for constants).

9.3.6 Multiplication of two positive numbers

If both multiples are known to be positive, multiplication can be achieved by simply changing the sign of the higher bound of one of the arguments followed by *_mm_mul_pd*. If one of the numbers is a constant, one can prepare it in a suitable way to avoid the sign change and implement the multiplication as a single instruction.

9.3.7 Division

Definition:

$$\frac{x}{y} = \left[\min\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right), \max\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right)\right],$$

undefined if $0 \in y$.

Again, this computation would require 8 divisions. Unfortunately, division is a rather slow operation, that is why we would prefer to use as few divisions as possible. One way to do this is to use $\frac{x}{y} = x\frac{1}{y}$, using the definition below, which uses only two divisions but quite a few other operations.

A more efficient (as well as more accurate) approach turns out to be the use of case distinction similar to (9.2). By examining the divisor, we end up with fewer possible cases and easy recognition of the exceptional cases. More specifically, the operation becomes:

$$\frac{x}{y} = \begin{cases} \left[\min\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}\right), \max\left(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right)\right], & \text{if} \quad 0 < \underline{y} \leq \overline{y} \\ \text{exception}, & \text{if} \quad \underline{y} \leq 0 \leq \overline{y} \\ \left[\min\left(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right), \max\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}\right)\right], & \text{if} \quad \underline{y} \leq \overline{y} < 0 \end{cases} \tag{9.3}$$

The final formula we use is

$$\frac{x}{y} \subseteq \left\langle \min\left(\nabla\left(\frac{a}{\underline{y}}\right), \nabla\left(\frac{-a}{(-\overline{y})}\right)\right), -\min\left(\nabla\left(\frac{-b}{(-\overline{y})}\right), \nabla\left(\frac{b}{\underline{y}}\right)\right)\right\rangle,$$

where

$$\begin{aligned} a &= \begin{cases} \underline{x} & \text{if } (-\overline{y}) \leq 0 \\ -(-\overline{x}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} (-\overline{x}) & \text{if } 0 \leq \underline{y} \\ -\underline{x} & \text{otherwise} \end{cases} \end{aligned}$$

with an additional check to throw an exception if $\underline{y} \leq 0 \leq \overline{y}$.

9.3.8 Reciprocal

Definition:

$$\frac{1}{x} = \left[\frac{1}{\overline{x}}, \frac{1}{\underline{x}}\right] \subseteq \left\langle \nabla\left(\frac{-1}{(-\overline{x})}\right), \nabla\left(\frac{-1}{\underline{x}}\right)\right\rangle,$$

undefined if $0 \in x$.

This is implemented as a check if the argument contains zero, followed by division of -1 by the argument and swapping the two components.

9.3.9 Absolute value

Definition:

$$|x| = [\max(\underline{x}, -\overline{x}, 0), \max(-\underline{x}, \overline{x})] = \langle \max(0, \underline{x}, (-\overline{x}))), -\min(\underline{x}, (-\overline{x}))\rangle\,.$$

9.3.10 Square

Implemented as $x^2 = |x||x|$, using multiplication of positive numbers.

9.3.11 Square root

Definition:

$$\sqrt{x} = \left[\sqrt{\underline{x}}, \sqrt{\overline{x}}\right]$$

only defined if $0 \leq \underline{x}$.

Since the rounding is an integral part of the square root operation, and in this case we cannot achieve a negated result, we need to use another method to ensure rounding in the correct direction. We use the fact already mentioned in the subsection on multiplication, $\Delta(r) \leq -\nabla(-\epsilon - \nabla(r))$.

The formula we use is:

$$\sqrt{x} \subseteq \begin{cases} \left\langle \nabla\left(\sqrt{\underline{x}}\right), -\nabla\left(\sqrt{-(-\overline{x})}\right) \right\rangle, & \text{if } \nabla\left(\nabla\left(\sqrt{-(-\overline{x})}\right)\right)^2 = -(-\overline{x}) \\ \left\langle \nabla\left(\sqrt{\underline{x}}\right), \nabla\left(\nabla\left(-\epsilon - \sqrt{-(-\overline{x})}\right)\right) \right\rangle, & \text{otherwise} \end{cases}$$

(where ϵ is the smallest representable positive number).

The condition for making the first choice in this formula is only satisfied if the result of $\sqrt{-(-\overline{x})}$ is exactly representable, in which case $\nabla\left(\sqrt{-(-\overline{x})}\right) = \Delta\left(\sqrt{-(-\overline{x})}\right)$. Otherwise the second choice adjusts the high bound to the next representable number.

Note that if we don't require tight bounds, using only the second choice in the equation above is sufficient to implement interval square root.

If the argument is entirely negative, the implementation will raise an exception. If it contains a negative part, the implementation will crop it to only its non-negative part, to allow that computations such as $\sqrt{0}$ can be carried on in exact real arithmetic.

9.4 Transcendental functions

If the implementations of transcendental functions in the standard C/C++ libraries satisfied the requirements of IEEE-754 rounding, interval versions of them could be implemented in a manner similar to above. Unfortunately, the accuracy of these libraries (or hardware implementations) is notoriously unreliable. Moreover, it is almost never possible to find information about the error bounds of these functions, which vary from architecture to architecture and even with different compilers and different compiler versions on the same machine.

Thus we decided to implement transcendental functions on intervals that produce certified bounds enclosing the result. They do not try to give tight (correctly rounded) bounds, instead prefer to overestimate but compute quickly. The elaborate theory and complicated implementation required to give tight bounds are beyond the scope of the intended application of our interval arithmetic implementation.

All the implementations rely on polynomial approximations generated using an implementation of the Remez algorithm (see e.g. [10]) with exact computation and certified error bounds. However, instead of finding the best Chebyshev approximation and using interval coefficients containing the real ones, we use multi-step approximation (suggested by [31]) where we approximate, round the highest-order coefficient to a double-precision number and then approximate again with a lower-degree polynomial using this rounded value as a fixed coefficient. The final coefficient is taken as an interval, expanded to accommodate the approximation error and rounded outwards.

With this the approximation of the function in its primary interval only requires the computation of this polynomial with interval arithmetic (in fact we do a little bit better, discussed below). The fact that all coefficients except the final additive are double-precision numbers helps to reduce the growth of the intervals. We choose our primary intervals to contain only non-negative numbers, so that multiplication of intervals can be performed as the special case that requires only two multiplications in a single instruction. For an additional speed-up, the polynomial evaluation is done using Estrin's algorithm (see [52]) to maximize parallelism.

For a monotone function, we know that if $P(x) = c_n x^n + ... + c_1 x + c_0$ chosen so that $P(x) - e \leq f(x) \leq P(x) + e$ for all x in some non-negative interval $[a, b]$,

$$\begin{aligned} P(\underline{x}) - e &\leq f(\underline{x}) &&\leq P(\underline{x}) + e \\ P(\overline{x}) - e &\leq f(\overline{x}) &&\leq P(\overline{x}) + e \end{aligned}$$

but for any $x \in [\underline{x}, \overline{x}]$, $f(\underline{x}) \leq f(x) \leq f(\overline{x})$, thus

$$\underline{P}(\underline{x}) \leq P(\underline{x}) - e \leq f(x) \leq P(\overline{x}) + e \leq \overline{P}(\overline{x}),$$

i.e. $f[x] \subseteq \left[\underline{P}(\underline{x}), \overline{P}(\overline{x})\right]$, where $\underline{P}$ (and $\overline{P}$) is P computed in such a way that it gives a lower bound for $P(x) - e$ (resp. a higher bound for $P(x) + e$). If all the coefficients are positive, this can be accomplished by simply rounding all coefficients down (resp. up), with the exception of c_0, which would also have to accommodate e, i.e. $\underline{c_0} = \nabla(c_0 - e)$ (resp. $\overline{c_0} = \Delta(c_0 + e)$). In our special case where all coefficients except c_0 are exactly represented in double precision, the coefficients of $\underline{P}$ and $\overline{P}$ coincide with the coefficients of P except for the very last one, c_0.

Unfortunately, in the presence of inexact operations, the evaluation of the polynomial is not so easy to do if the coefficients are not all positive. A negative coefficient requires an upper bound for x^i, which would be a nuisance to compute and would add up to the uncertainty of the result. However, in the cases we actually use we have patterns that can be exploited, e.g. alternation between positive and negative coefficient. In the latter case, in the computation of $\underline{P}$ we can assume $\underline{x}$ is given exactly, thus we can compute pairs $c_{i+1}x + c_i$ rounded in the correct direction (these pairs are actually required by Estrin's algorithm). If we, additionally, know that all these pairs are positive (e.g. if $0 < x \leq 1$ and $-c_{i+1} \leq c_i$), the computation can proceed from there using only lower bounds for the even powers of x.

The transcendental functions in the current implementation of the interval arithmetic package of *RealLib* are not very precise, i.e. they overestimate the output intervals. The main reason for this is the use of Estrin's algorithm, which was chosen for its superior performance. At the moment we are considering improving the accuracy of these functions whenever such improvements would not drastically influence their performance.

On the other hand, since the functions do provide correct enclosures in very little time, and overestimation is one of the principles on which the exact real number library *RealLib* is based, the current transcendental functions completely serve their purpose as part of the library.

9.4.1 Sine and cosine

Sine and cosine are non-monotonic functions, which means that one cannot simply use $\sin x = [\sin \underline{x}, \sin \overline{x}]$. Instead, we use the fact that both functions are non-expansive and thus

$$\sin x = \left[\sin\left[\frac{\underline{x}+\overline{x}}{2}\right] + \frac{\underline{x}-\overline{x}}{2}, \sin\left[\frac{\underline{x}+\overline{x}}{2}\right] + \frac{\overline{x}-\underline{x}}{2}\right],$$

where by $\sin[x]$ we mean evaluation of sin on the interval $[x, x]$, returning an interval containing the result.

The latter we compute by a polynomial approximation of the function $\sin \frac{x}{3}$ on the interval $[-\pi, \pi]$ by an 8-coefficient polynomial, such that $\sin \frac{x}{3} \approx xP(x^2)$. Before we can apply this, we use range reduction (which can be safely performed as x is a real number and not an interval) to make sure $x \in [-\pi, \pi]$. To get the final value of $\sin x$, we use the identity $\sin(3x) = (3\sin^2 x - 4)\sin x$.

The computation of cosine is done in a very similar manner, the only significant difference is that the approximation used is $\cos \frac{x}{3} \approx P(x^2)$, i.e. the computation requires one multiplication less.

9.4.2 Arctangent

Two versions of this function are used in practice, one is the simple arctangent and the other one takes two arguments and gives a result that depends on the signs of both of them so that it can be directly used to compute polar coordinates or arguments of complex numbers.

Both are implemented using case distinctions and a common function that computes the arctangent for the primary interval $[0, 1]$. For the cases that contain numbers on the boundaries (e.g. $[-0.9, 1.1]$), we use the fact that arctangent is lipschitz continuous with constant one, this we simply return a constant interval expanded to accommodate the width of the input interval.

The computation on the primary interval is done simply by a polynomial approximation with 20 coefficients of alternating sign.

9.4.3 Exponent

Since the floating point representation of numbers uses a base-2 exponent, the easiest way to perform exponentiation is to transform the argument to base

2 (i.e. simply multiply by $\log_2 e$), separate the integer and fractional part, use some bit operations to construct a number with the integer part as the exponent, approximate the exponent of the fractional part and combine the two components using multiplication.

We do this separately for the lower and upper bounds of the interval. The extraction of integer and fractional part is an exact operation, but any other step in the computation requires that we round in the appropriate direction. The 12-coefficient polynomial approximation of 2^x for $x \in [0, 1)$ we use contains only positive coefficients, thus it presents no problem. The initial and final multiplication are done according to the rules of interval multiplication with one (resp. two) positive multiples.

9.4.4 Logarithm

The range reduction in the logarithm case is the inverse of the work done for exponentiation, with a few additional steps.

The mantissa and exponent are separated using a few bit operations, to produce a mantissa in the range $[0.5, 1)$. Unfortunately the direct approximation of the function $\ln x$ on this interval does not give us a polynomial which can be safely evaluated separately for the lower and upper bounds of an interval.

Instead, we approximate $\ln(1 - x)$ where $x \in (0, 1 - 2^{-\frac{1}{4}}]$, using two steps of range reduction to limit the number of coefficients to 14 (all positive). The range reduction is accomplished by choosing x or $x2^{2^{-i}}$ (for $i = 1, 2$) depending on whether the latter is smaller than one, adjusting the exponent by adding 2^{-i} if that is the case.

The result of the polynomial approximation is finally added to the (adjusted) exponent, multiplied by $\ln 2$.

9.5 Performance

We compare the performance of this implementation to the performance of two other packages for interval arithmetic freely available on the internet: the interval part of the *Boost* project (version 1.33.0, [115]) and the library *filib++* (version 2.0, [40]). For the latter, we tried the macro version as well as two of the available rounding policies, *multiplicative* and *native_onesided_global*, the latter corresponding most closely to our method of rounding.

The results of the benchmark are summarized in the following table, showing the ratio between the performance of the respective library and double precision floating point:

operation	*filib++*, macro	*filib++*, onesided	*filib++*, multiplicative	*Boost*	*RealLib3*
$+$	6.52	2.64	6.84	10.72	1.11
$*$	7.86	3.40	7.93	113.47	5.50
$/$	12.42	3.98	10.06	9.60	3.80
$\sqrt{\cdot}$	25.43	63.97	63.17	16.54	2.00
$\lvert\cdot\rvert$	27.11	20.23	20.21	1.61	2.62
sin	2.91	2.63	2.73	-	0.63
cos	2.92	2.75	2.74	-	0.58
arctan	2.26	6.20	6.42	-	1.27
$e^{\cdot}$	3.45	22.25	40.30	-	0.86
ln	3.86	5.91	6.13	-	0.94
$\sum_{i=1}^{1000000} \frac{1}{i}$	3.19	1.51	2.74	4.80	1.53

(Pentium-M 1.8GHz, Windows XP + Cygwin, GCC 3.4.4)

Several cells are blank, because *Boost* does not provide transcendental operations.

RealLib is faster almost everywhere, with the notable exception of multiplication in *filib++*'s *native_onesided_global* mode. In this case *filib++* uses a case distinction, which in our test only reaches the shortest of the 9 possible paths. We prefer not to explore the performance of *filib++*'s multiplication in cases where the signs of the arguments change in an unpredictable manner. Our implementation does not have such a problem as it only uses one execution path for all multiplications, thus the ratio given in the table is both best and worst case performance.

9.6 Intel's SSE3

The latest multimedia extension set introduced by Intel, the SSE3 [120], aimed at improving complex number computations, does not provide any benefit for interval computations. Intel chose to improve complex multiplications and divisions by introducing the instruction *_mm_addsub_pd*, which combines two packed registers by adding one of the two components and subtracting the other [119]. Unfortunately, the use of this instruction leads to incorrect results if a directed rounding mode is in effect, because the multiplication that precedes the substraction is rounded in the wrong direction.

A better handling of complex multiplications would have been the introduction of a multiplication instruction "*mulpn*" (for multiply positive negative) that changes the sign of one of the components of one of the arguments. This would require the same effort that the instruction *_mm_addsub_pd* required, but would have the correct behavior in directed rounding modes, i.e. complex multiplication code using *mulpn* would yield upper bounds for the result of the multiplication if rounding towards $+\infty$ is in effect, and lower bounds in the case of rounding towards $-\infty$.

Unlike *_mm_addsub_pd*, a *mulpn* instruction would have been useful and advantageous for interval arithmetic. Multiplication of two positive numbers could be implemented as a single *mulpn*, which would also speed up the implementa-

tions of transcendental interval functions.

9.7 Suggestions for a hardware implementation

We hope that the presentation until this point has convinced the reader that the use of the storage $\langle \underline{x}, -\overline{x} \rangle$ for intervals in SSE2 registers is clearly superior to the traditional method of storing intervals as simply the pair of the two bounds. This mode of storage avoids the need for special rounding modes in a hardware implementation, and even turns some existing instructions into meaningful interval operations.

We propose this storage to be adopted as the preferred storage format for intervals in hardware implementations.

To further speed up computations on intervals, we propose the introduction of a special selection instruction we call *ivchoice* (for interval choice) that can be used to prepare the arguments for multiplication and division. The action of this instruction should correspond to the following function:

```
 __m128d ivchoice(__m128d a, __m128d b) {
    a = _mm_xor_pd(a, _mm_set_pd(0.0, -0.0));
    a = _mm_shuffle_pd(a, a, _mm_movemask_pd(b));
    return a;
}
```

This is pseudocode, because *_mm_shuffle_pd* cannot be performed based on a non-const integer. A software implementation of the above requires a switch statement, which can slow the execution considerably, especially in cases where the signs of the multiples cannot be predicted.

If such an instruction is available, the multiplication algorithm becomes:

```
__m128d IntervalMul(__m128d x, __m128d y) {
    __m128d a = _mm_shuffle_pd(x, x, 1);
    __m128d b = _mm_shuffle_pd(y, y, 1);
    __m128d c = ivchoice(b, x);
    __m128d d = ivchoice(y, a);
    __m128d e = _mm_mul_pd(c, x);
    __m128d f = _mm_mul_pd(d, a);
    __m128d g = _mm_min_pd(e, f);
    return g;
}
```

If the latency of the proposed instruction can be the same as the latency of *_mm_shuffle_pd*, this sequence of instructions will run about 30% faster than the current implementation.

Moreover, since the multiplications above only use the results of *ivchoice* with the same second argument, it is even possible to fuse *ivchoice* with the multiplication that is applied to the result. The extent to which such fusion can be beneficial depends on the actual hardware implementation. If the latency of *ivchoice* can be folded completely (which seems possible) or partially, interval multiplication using the fused "*ivmul*" could reach a latency close to the latency of two dependant double precision multiplications.

Apart from an additional test if the divisor contains zero and the use of *_mm_div_pd* instead of *_mm_mul_pd*, the division code is identical to the multiplication one:

```
__m128d IntervalDiv(__m128d y, __m128d x) {
    if (_mm_movemask_pd(x)==3)
        throw exception;
    __m128d a = _mm_shuffle_pd(x, x, 1);
    __m128d b = _mm_shuffle_pd(y, y, 1);
    __m128d c = ivchoice(b, x);
    __m128d d = ivchoice(y, a);
    __m128d e = _mm_div_pd(c, x);
    __m128d f = _mm_div_pd(d, a);
    __m128d g = _mm_min_pd(e, f);
    return g;
}
```

Fused *ivchoice* and division ("*ivdiv*") is also possible.

Of course, one would prefer to have a complete hardware implementation of interval arithmetic that provides instructions for the four basic operations on intervals. In our mode of operation addition already has a hardware implementation as a single instruction. Substraction would require a fusion of swapping and addition ("*ivsub*") which should be easy to accomplish in hardware without extra latency compared to addition.

On the other hand, multiplication and division seem too complex to be directly implemented. A pure hardware implementation of multiplication may be able to choose execution paths without the delays associated with incorrect branch predictions, thus probably the preferable hardware design would examine the signs of the four components to choose one of 9 possible combinations and perform a single pair of multiplications in 8 of the possible cases. In the 9'th case, however, the operation would require the same amount of work as the function *IntervalMul* above.

Since the worst-case latency would be the same as the algorithm above, the latter should not be ignored as a possible basis for a pure hardware implementation of interval multiplication.

To conclude, we suggest that hardware assistance for interval computations should be provided as the adoption of the $\langle \underline{x}, -\overline{x} \rangle$ storage format and the introduction of the instructions of one of the following three levels:

basic	*mulpn*, *ivsub*, *ivchoice*
advanced	*mulpn*, *ivsub*, *ivmul*, *ivdiv*
full	*ivsub*, *IntervalMul*, *IntervalDiv*

The advanced level seems to be the best combination of feasibility and performance.

9.8 Related work

In [35], von Gudenberg discusses the efficiency of implementations of interval arithmetic using the multimedia extensions Intel's SSE, AMD's 3DNow! and Motorola's AltiVec. The paper concludes that the use of multimedia extensions

only leads to a very modest improvement in multiplication with Intel's SSE in comparison to standard floating point, and only due to the fact that four single-precision operations can be executed in parallel.

Unlike SSE, the double precision second version of the extensions, SSE2, is a natural candidate for interval arithmetic because the packed registers hold two double precision values.

Von Gudenberg used a variety of rounding policies, the fastest of which is global onesided rounding, the method we use, but did not store one of the components negated in memory. Consequently, handling the negations required to perform rounding in the proper direction increases the number of instructions needed for every operation. If we were to use SSE2 in a similar mode of operation, the required number of instructions for addition would be four instead of one, for sign change – two instead of one, for substraction – five instead of two, and for multiplication of positive intervals – three instead of two.

Additionally, instead of 9-case branching on the signs of the 4 components, we prefer to use 4 multiplications with selected arguments (the selection is branch-free), which gives us stable performance that is not affected by branch mispredictions or longer latency execution paths, although with a slightly worse best-case performance.

In [70], Kolla, Vodopivec and von Gudenberg discuss the possibility of hardware extensions supporting interval arithmetic similar to the multimedia extensions 3DNow!, via packed storage of single precision numbers in a double precision register. For addition and substraction they require special instructions that round each component of the pair in the appropriate direction, and for multiplication they describe a case selection method that can easily be implemented and be very efficient for 8 of the 9 possible cases and requires a sequence of operations and longer latency for the (rare) 9'th case.

We are quite skeptical about the chances of such a complicated multiplication instruction ever being implemented in hardware. Instead, we give a much more modest proposal that can also lead to very good performance at the cost of little extra hardware. It also has the benefit that one of the operations, addition, already has a hardware implementation.

In [21], Ershov and Kashevarova report on implementations of transcendental functions, based on the Chebyshev and Taylor approximations of these functions. They note that three sources of error have to be accounted for in the computation of approximating polynomials:

- the error caused by finitely approximating an infinite sequence,
- the error in the approximation of the coefficients of the polynomial,
- the error caused by inexact operations.

The use of rounded coefficients influencing the choice of approximating polynomials in our approach nearly invalidates the need to consider the second source of error above. Our approximating polynomials only contain coefficients that are correctly representable as double precision floating point numbers, with the exception of the first coefficient, whose interval representation could be modified so that it also covers the first source of error in the list above.

Appendix A

Introduction to the **RealLib** library for exact real arithmetic

A.1 The real numbers interface

The class *Real* is the main class in the system. It contains the description of a real number and can be used to extract properties of this real number, as well as to apply operations to it.

A real number can be constructed in several different ways:

- from a constant entered as a *double*, taken to be exact, e.g. *Real*(1.5) would define the number 1.5 exactly, while *Real*(0.1) would define the real number which matches the double precision representation of 0.1, which is different from 0.1 by about $5.55 \cdot 10^{-18}$;
- from a string, e.g. *Real*("0.1") would define the number 0.1 exactly;
- via operations applied to *Real* arguments, e.g. *Real*(1)/*Real*(3) will define $\frac{1}{3}$ exactly;
- via functions applied to *Real* arguments, e.g. *sqrt*(*Real*(2)) for $\sqrt{2}$;
- using a predefined (*Pi* and *Ln2*) real constant or one be created using the interfaces described in Section A.2;
- from oracle functions, i.e. functions that can return arbitrarily good decimal approximations to a number.

The use the term "constructing a real number" instead of "assigning a value to a real variable" here is intentional. Real numbers are represented via structures that describe the computation through which they were computed. Every time a new operation is performed, a new object is being created that describes the operation and contains references to the objects that were arguments to the operation. In this sense, updating the value of a variable of type *Real* usually does not mean that the objects that described the earlier value are destroyed. The latter only happens if the variable has not been used in other operations.

The main purpose of the real numbers layer is to be able to extract properties of a number that is constructed. These can be:

- an approximation to the value in *double* precision, different from the actual value by at most 1 ulp[1];
- a decimal representation of the real number, which is at most different by 1 in the least significant position[2];
- strict comparisons, which will loop indefinitely if the numbers are equal. It is impossible to test two real numbers for equality, thus the system does not provide the operators ==, <= or >=.

Let us take a closer look at how the system behaves with a few simple program fragments:

[manual/fragments.cpp]

```
001 #include <iostream>
002 #include <iomanip>
003 #include "Real.h"
004 using namespace std;
005 using namespace RealLib;
006
007 void main() {
008    InitializeRealLib();
009    Real a, b(4);
010    b = Pi / b;
011    a = sin(b);
012    cout << "sin(Pi/4) is " << a << endl;
...
031    FinalizeRealLib();
032    return 0;
033 }
```

Lines 1 to 5 include the necessary headers, and lines 8 and 31 perform the initialization and finalization of the library.

Line 9 declares two variables of type *Real*. One of them is initialized to the default value (which is 0), the other to the constant 4. Line 10 updates *b* with the result of the division of the constant *Pi* and *b*. If we look at this with more detail, the system constructs a new real number which is the result of the application of the operation to its arguments (in this case $\frac{\pi}{4}$), keeping a reference to the constant *Pi* and the object that *b* was assigned to, after which it removes the link between *b* and the constant 4 object. Since this object is still needed as argument to the division operator, it will not be deleted.

Line 11 updates *a* with the result ($\sin\frac{\pi}{4} = \frac{\sqrt{2}}{2}$) of the function *sin* applied to the argument *b*. That is, the system creates a new object that describes the operation "*sin* applied to the object that *b* links to", and links *a* to it. The link is not to the variable *b*, but to the object that it linked, i.e. the real number it held, $\frac{\pi}{4}$. The previous value of *a* (the zero it was implicitly initialized to) was not used anywhere, thus it is not needed any more and will be deleted by the system.

[1] A correctly rounded approximation according to the IEEE-754 standard is not possible to achieve because of the undecidability of the equality test for real numbers.

[2] Correct rounding is impossible to achieve. This may mean all digits are incorrect: in different scenarios the system may print either of 1.000 and 0.999 for the real number 0.9995.

Line 12 prints out the result. In order to do this, the system will run through the objects linked to *a* and generate an approximation that can be printed on the screen. This approximation will be cached for possible later use. This fragment prints

```
sin(Pi/4) is 0.707107
```

The next fragment demonstrates a slightly more complicated case:

```
...
014    Real c(a * 2 / sqrt(Real(2)));
015    for (int i=0; i<1000; ++i)
016       c = sin(c);
017    cout << "sin(sin(...(1)...)) (1000 times) is " << c << endl;
...
```

Line 14 declares *c* to be 1 using the fact that the value of *b* is equal to $\frac{\sqrt{2}}{2}$. In a usual floating point environment such a roundabout definition would certainly introduce a significant accuracy loss. This is not the case here: performance may suffer, but the accuracy will still be full.

Lines 15 and 16 run a loop in which *sin* is consecutively applied to *c* 1000 times. In reality, this means that every run through Line 16 a new object is created that describes an application of the function *sin* to the previous object. In the end of this loop, *c* will point to a structure that looks like this:

$$c \to sin \to sin \to \ldots \to operator\ / \to \begin{cases} operator\ * \to \begin{cases} sin \to operator\ / \to \begin{cases} Pi \\ 4 \end{cases} \\ 2 \end{cases} \\ sqrt \to 2 \end{cases}$$

At Line 17, this structure is traversed (using the cached value of $\sin \frac{\pi}{4}$) to construct an approximation to the number which is good enough to be displayed:

```
sin(sin(...(1)...)) (1000 times) is 0.054593
```

The necessity for the structure comes from the possibility that the approximation may be not good enough to ensure that the requested number of digits be displayed. This will happen when we run through the next line, which requests 120 decimal digits of accuracy, well beyond the accuracy of the default initial precision of the system:

```
...
019   cout << " or " << scientific << setprecision(120) << c << endl;
...
```

Here the system will check if the cached value of *c* (computed in Line 17) is good enough for the new request. Since it isn't, this will cause the system to clear all cached values and run through the description *c* points to again with higher precision in order to get better accuracy. The new precision may again be insufficient, which will trigger another iteration and this process will continue

until the accuracy is good enough or the maximum precision[3] is reached[4]. As a result, we see

```
 or 5.459297151018517740315360672399401040105753696174101612835207082077700697973467364856621073573888729604355226314181717780e-2
```

Line 19, along with the following lines, also demonstrates some of the formatting manipulators that the system's output functions respect.

```
...
021    cout << "the double representation of 0.1 is "
022       << fixed << noshowpoint << Real(0.1) << endl;
023    cout << "and its distance from 0.1 is "
024       << showpos << scientific << uppercase
025       << showpoint << Real(0.1) - Real("0.1") << endl;
...
```

The output of this fragment:

```
the double representation of 0.1 is 0.1000000000000000055511151231257827021181583404541015625000000000000000000000000000000000000000000000000000000000000000000000000000000
and its distance from 0.1 is
+5.551115123125782702118158340454101562500000000000000000000000000000000000000000000000000000000000000000000000000000000E-18
```

The other ways to extract properties of the real number are shown in the next few lines, where you can also see that *double* cannot even print its own value correctly:

```
...
026    cout << "double(0.1) < Real(\"0.1\"): "
027       << boolalpha << (Real(0.1) < Real("0.1")) << endl;
028    cout << "0.1 converted to double is "
029       << noshowpos << fixed << Real("0.1").AsDouble() << endl;
...
```

The output of this fragment:

```
double(0.1) < Real("0.1"): false
0.1 converted to double is 0.1000000000000000055511151231257827021181583404541015625000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
```

The following is a complete program written using only the real numbers interface to compute the sum of the harmonic series for 1000000 members. We have added some code to track the time it takes for every stage of the computation to complete.

[manual/harm.cpp]

```
001 #include <iostream>
002 #include <iomanip>
003 #include <ctime>
004 #include "Real.h"
```

[3] specified as an argument of *InitializeRealLib*(), see Section B.1 for details

[4] which will cause an exception that can be caught by the user, see Section B.1

```
005
006 using namespace RealLib;
007 using namespace std;
008
009 #define LENGTH 1000000
010 #define MACROTOSTRING2(x) # x
011 #define MACROTOSTRING(x) MACROTOSTRING2(x)
012
013
014 Real Harmonic(const int mcnt)
015 {
016    Real one(1);
017    Real s;   // initialized to 0
018    for (int i=1; i<=mcnt; ++i) {
019       s += one/i;
020    }
021    return s;
022 }
023
024
025
026 int main()
027 {
028    clock_t starttime, endtime;
029
030    cout << "Computing the sum for " MACROTOSTRING(LENGTH) " members"
<< endl;
031
032    starttime = clock();
033    InitializeRealLib();
034    {
035       Real h(Harmonic(LENGTH));
036       endtime = clock();
037       cout << "construction time: " <<
038          double(endtime - starttime) / CLOCKS_PER_SEC << endl;
039
040       for (int n=10; n<500; n*=6) {
041          starttime = clock();
042          cout << unitbuf << fixed << setprecision(n);
043          cout << n <<" digits: \t" << h << endl;
044          endtime = clock();
045          cout << fixed << setprecision(6);
046          cout << "prec: " << GetCurrentPrecision() << " time elapsed: "
<<
047             double(endtime - starttime) / CLOCKS_PER_SEC << endl;
048       }
049       starttime = clock();
050    }
051    FinalizeRealLib();
052    endtime = clock();
053    cout << "destruction time: " <<
054       double(endtime - starttime) / CLOCKS_PER_SEC << endl;
055
```

```
056
057    return 0;
058 }
059
```

The *clock()* pairs surround the regions of code that do the interesting work. First we measure the time it takes to initialize the system and to construct the representation of *h*, the real number that represents the 1000000-member sum, then we consecutively time the extraction of representations with different number of decimal digits, and finally we measure the time needed to destroy the representation when it goes out of scope. The extraction is the place where the actual computation is performed, and the accuracies are chosen to require a single new iteration through the representation.

The result of the execution[5] of this program:

```
Computing the sum for 1000000 members
construction time: 3.845
9 digits:   14.3927267
prec: 4 time elapsed: 24.806
63 digits:  14.392726722865723631381127493188587676644800013744311
1653418433
prec: 9 time elapsed: 20.049
441 digits:     14.392726722865723631381127493188587676644800013744
443116534184330458129585075179950035682981759472191007083599521360
079812900264164102586930094633006200549611666639142755843266541577
219730782928819514121133122033133043828972712951321469882948594557
104755079764875032609612144070163003538369161116798217677091946824
417163326372248859422898758102848526351896600065279756908532436955
245532742791258943257193916658973962848216357840564467417355069077
586
prec: 48 time elapsed: 23.463
destruction time: 0.942
```

As you can see from the timings, the computation time did not change much when we went from the initial precision to 9 32-bit words, and when we more than quintupled the precision for the 441-digit approximation. This shows that something is wrong, i.e. that too much time is being spent somewhere else, not in performing the actual computation.

When the class *Real* is used to perform computations, the real numbers are represented as functions and the system acts as a type-2 machine to transform functions into functions. This is the traditional approach for real number computations, which suffers from serious efficiency problems (see Chapter 8).

To deal with these efficiency problems, the system offers the real functions interface where the user can create functions that work on the more efficient approximations level. In the next section we will see how this program can be changed to make use of this and obtain a dramatic performance improvement.

[5] machine used: Pentium M 1.8GHz, 2MB Level-2 cache, 512 MB DDR-333 main memory, GCC 3.3.3 in Cygwin environment

A.2 The real functions interface

The function *Harmonic* is the only part of the program we need to change. In particular, Lines 13-23 change to the following:

[manual/harmfun.cpp]

```
013 template <class TYPE>
014 TYPE Harmonic(const long prec, const long mcnt)
015 {
016    TYPE one(1);
017    TYPE s;   // initialized to 0
018    for (int i=1; i<=mcnt; ++i) {
019       s += one/i;
020    }
021    return s;
022 }
023 CreateIntRealFunction(Harmonic);
```

When we execute this program, we see identical approximations, but the timings differ significantly:

```
Computing the sum for 1000000 members
construction time: 0
9 digits:   14.3927267
prec: 4 time elapsed: 0.17
63 digits:  14.392726722865723631381127493188587676644800013744311
1653418433
prec: 9 time elapsed: 6.079
441 digits:     14.392726722865723631381127493188587676644800013744
4431165341843304581295850751799500356829817594721910070835995213
0798129002641641025869300946330062005496116666391427558432665415
7
2197307829288195141211331220331330438289727129513214698829485945
5
1047550797648750326096121440701630035383691611167982176770919468
2
4171633263722488594228987581028485263518966000652797569085324369
5
2455327427912589432571939166589739628482163578405644674173550690
7
586
prec: 48 time elapsed: 9.363
destruction time: 0
```

You can see that the time it takes to obtain the more accurate approximations was reduced at least in half. The more important difference is the change in the time it takes to compute the least accurate approximation: instead of more than 20 seconds, this computation now takes 170 milliseconds! This is significantly less than even the time the original program needs to construct or destroy the number's representation.

In fact, when the machine precision is sufficient for the computation, this modified program runs at a speed in the order of magnitude of an identical computation implemented in *double*. While it is not easy to evaluate the accuracy of the latter, the results obtained using *RealLib* can be trusted, and the cost of obtaining them is not as dramatically different as it is with earlier exact real number systems[6].

[6] A comparison of *RealLib* with other packages and hardware floating point is given in Chapter 8.

The functions layer in *RealLib* avoids the complex processing needed to construct term descriptions of the real numbers by using the user's code as the description. In order to do this, the user needs to extract the bulk of the computation into a function that is written in a way appropriate for the function layer of the system.

Let us look with more detail at the changes we needed to apply to use the real functions interface:

- the function *Harmonic* is now a function template;
- its signature is changed to accommodate one extra argument (precision) which is not used in the function;
- the *CreateIntRealFunction* macro is used to create a mapping on the level of real numbers linked to that function.

To make fully efficient use of specialized precision code, the system requires the user's functions to be defined as function templates. The template gets instantiated for two[7] different approximation classes that share the same interface, described in detail in Section B.3. Defining the user's function as a template allows the system to make full use of the compiler's optimization abilities, especially in the very fast machine precision stage of the computation.

To be able to compute transcendental functions or numbers, the user functions are supplied with an additional argument that specifies the precision the system expects from the user's function. The programmer can use this parameter to decide e.g. the length of the series that approximates a number. A use of this will be shown later– our current example ignores this parameter as it computes the needed value exactly[8].

Our function is a function that takes one integer argument and returns a real number. In the system such functions are called "nullary real functions" (as they do not take a real argument), and are declared using this signature:

```
template < class TYPE >
TYPE name(unsigned int precision, UserInt userarg)
```

And finally, a function written for the function interface is not very useful unless it has a representation on the level of the real numbers, where it can be applied and its results can be examined. This representation is created by the linking macro, in this case *CreateIntRealFunction*, which maps a nullary real function into a function of this signature:

```
Real name(UserInt arg)
```

In the rest of the program, this function is used in the same manner as the original version written on the real numbers layer.

Let us now look at the implementation of a transcendental function that takes one real argument:

[7] in the current version, the number may change in the future

[8] The theoretical model used for the system allows one to define real number functions that operate on the level of approximations and are modular on that level; since all built-in and used-defined functions follow the requirements of the model, the computations can be assumed to be carried out exactly; although in reality they only provide approximations, the error in these approximations is accounted for and should be ignored by the user.

[manual/exp.cpp]

```
001 #include <iostream>
002 #include <iomanip>
003 #include "Real.h"
004
005 using namespace RealLib;
006 using namespace std;
007
008 template <class TYPE>
009 TYPE myexp(const TYPE &arg)
010 {
011    unsigned int prec = arg.GetPrecision();
012    TYPE s(0.0);
013    TYPE m(1.0);
014
015    if (abs(arg) > 1.0) throw DomainException("myexp");
016    if (!(abs(arg) < 1.0)) throw PrecisionException("myexp");
017
018    for (unsigned i=1; i<=prec; ++i) {
019       s += m;
020       m = m*arg/i;
021    }
022    return s.AddError(m*3);
023 }
024 CreateUnaryRealFunction(myexp)
025
026 int main()
027 {
028    InitializeRealLib();
029    {
030       Real a(myexp(Real(0.5)));
031       Real b(exp(Real(1)));
032
033       cout << fixed << setprecision(10);
034       cout << "a(myexp(0.5)) =\t" << a << endl;
035       cout << "a*a =\t\t" << a*a << endl;
036       cout << "b(exp(1)) =\t" << b << endl;
037       cout << "a*a/b =\t\t" << setprecision(300) << showpoint
038          << a*a/b << endl;
039    }
040    cout << "precision used: " << FinalizeRealLib();
041
042    return 0;
043 }
044
```

The template function *myexp* defined on Lines 8-23 is the definition of a function that computes the exponent of numbers in the range $(-1, 1)$.

At Line 11 we extract the precision, given as the approximate number of correct 32-bit words that we need to achieve from the argument[9]. For this first

[9]Theoretically this argument allows the construction of a sequence converging to the result of the application of the function. To achieve the best performance, functions should try to

attempts, we will not try to match this request, but will only make sure that our approximation's accuracy increases as *prec* increases.

Line 15 makes sure we reject arguments that are clearly outside the domain of our function. If the argument is on the boundary, we cannot recognize that and reject it, but we can wait until the argument is inside our domain. Since our domain is an open interval, to any real number in it there will always be an approximating interval that fits inside the domain (when the precision is high enough). Line 16 makes sure we do not compute the result until we have such an approximation.

After this, Lines 18-21 compute the Maclaurin expansion of the exponent function with *prec* members, and Line 22 adds an error that covers the remainder sum, provided that the argument is within $(-1, 1)$, which lines 15 and 16 have ensured.

Line 24 creates a mapping for the function on the level of real numbers that is in the form

```
Real myexp(const Real& arg);
```

When the main function creates a variable of type *Real* initialized to *myexp*(0.5) on Line 30, the system constructs an object that describes an application of the template function *myexp* to an object that describes the real number 0.5. There is no need for the creation of objects describing the steps of the computation of the Maclaurin sequence: this description is already present as the code of the template function *myexp*.

When the value of *a* is requested on Line 34, the library creates an approximation to the argument and runs an instantiation of the template function *myexp* to it. It then examines the error in the approximation the function returned. If the error is small enough to allow the display of a 10-digit representation, this representation is printed.

If it is not, the system will create a better approximation to the argument and will pass it to *myexp* (possibly a different instantiation of it), which will give a better approximation to the end result, because the parameter *prec* has increased and the argument is bounded in absolute value by 1. Eventually an approximation that is good enough will be obtained and will be printed.

Here is the result of the execution of this program:

```
a(myexp(0.5)) = 1.6487212707
a*a =       2.7182818285
b(exp(1)) = 2.7182818285
a*a/b =     1.000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
precision used: 272
```

Let us extend this example a bit. First, since the remainder bound $3m$ (which is an upper bound for me) on Line 22 is also valid for the real numbers

make this a rapidly converging sequence by aiming for an accuracy close to the requested. Unlike other libraries, the correctness of the results does not depend on the whether or not they succeed as long as the approximation errors are indicated as shown in this section.

1 and -1, we would like to expand our domain to $[-1, 1]$ to include them. Also, we want the best possible performance, thus we will try to satisfy the precision request. To do so, we will evaluate so many members of the sequence that $me \leq 2^{-32prec}$. We have two choices:

- pre-compute a bound for the number of terms needed and run the series this number of times. This will give the best performance for lower precisions, but may require expensive extra computations for higher precisions;
- run the series until the condition is satisfied. This implies extra computations for each iteration, but may be significantly faster in higher precisions if the argument is smaller.

We will use both, choosing by examining the value of the argument. We will use the inequality $\left(\frac{n}{3}\right)^n < n! < \left(\frac{n}{2}\right)^n$ (for $n \geq 6$) to get a the following upper bound for the number of terms:

$$n(\ln n - ln3) > 32prec \ln 2 + 1,$$

or very roughly $n \geq 23prec$ terms will be sufficient.

The modified function follows (the rest of the program does not change):

[manual/expimpr.cpp]

```
008 template <class TYPE>
009 TYPE myexp(const TYPE &a)
010 {
011    unsigned int prec = a.GetPrecision();
012    TYPE s(0.0);
013    TYPE m(1.0);
014
015    TYPE arg(a.TruncateTo(-1.0, 1.0, "myexp"));
016
017    if (abs(arg).weak_le(0.75)) {
018       TYPE err = (TYPE(1) >> (32 * prec)) / 3;
019       for (unsigned i=1; abs(m) > err; ++i) {
020          s += m;
021          m = m*arg/i;
022       }
023    } else {
024       if (prec < 6) prec = 6;
025       unsigned int pc = prec * 23;
026       for (unsigned i=1; i<=pc; ++i) {
027          s += m;
028          m = m*arg/i;
029       }
030    }
031    return s.AddError(abs(m)*3);
032 }
033 CreateUnaryRealFunction(myexp)
```

This code fragment demonstrates the following:

- how we can satisfy the precision request by choosing a suitable length for the approximating sequence;

- how we can use the member function *TruncateTo* to define functions that have a closed domain;
- how we can use a weak comparison to choose between different control paths that lead to the same result but may have different performance.

Here is the result of running this example file:

```
a(myexp(0.5)) = 1.6487212707
a*a =       2.7182818285
b(exp(1)) = 2.7182818285
a*a/b =     1.000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
precision used: 34
```

The precision that is needed to compute the ratio is now very close to optimal, and thus the computation itself runs much faster.

Appendix B

Reference of the classes and functions of RealLib

All of the library's functions, constants and operators live in the *RealLib* namespace and are included using the header `Real.h`.

B.1 Initialization and finalization, exceptions

The system must be initialized prior to use and finalized afterwards.

```
void InitializeRealLib(
   unsigned precStart = MachineEstimatePrecision,
   unsigned precMax = 100000,
   unsigned numEstAtStart = 1000);
```

Initializes the library. The starting precision is specified in the first argument. If nothing is specified, the system will start with machine precision floating point approximations.

precMax specifies the maximum working precision. If the system cannot decide a property after reaching this maximum precision, it will abort with a *PrecisionException* that can be caught by the user.

numEstAtStart specifies the amount of space the library will reserve for approximations at initialization. If more space is needed, the library will increase the storage appropriately. In such a case, specifying a higher *numEstAtStart* may save a few memory reallocations.

```
#define MachineEstimatePrecision 4
```

A value that is used to indicate interval arithmetic with double precision. This is the default initial precision in the system.

```
unsigned FinalizeRealLib();
```

Finalizes the library. All cached approximations are destroyed, the memory allocated is freed and the current precision is returned.

```
unsigned ResetRealLib(
   unsigned precStart);
```

Resets the library, setting a new working precision. Useful when one com-

putation is complete, and another must start from the initial precision to avoid working with unnecessarily high precision.

```
unsigned GetCurrentPrecision();
```

Returns the current precision in 32-bit words. A value of *MachineEstimatePrecision* means that the system is currently working with interval arithmetic with double precision.

```
class RealLibException : public std::exception {
   char m_what[128];
public:
   RealLibException(const char *what = NULL) throw();
   virtual const char *what() const throw();
};
```

Base class for the exception classes used in the system. The constructor takes one string argument specifying a text message for the place the exception originated, and the *what* member function returns a pointer to this string. The following two classes share this interface:

```
class PrecisionException : public RealLibException {
public:
   PrecisionException(const char *what = NULL) throw();
};
```

Raised by functions when the current approximation was not sufficient to know anything about the resulting approximation. Indicates the system must start a new iteration with higher precision.

If this exception is passed on to the user, this means that the maximum precision specified in *InitializeRealLib* has been reached and was not sufficient to extract the wanted property.

```
class DomainException : public RealLibException {
public:
   DomainException(const char *what = NULL) throw();
};
```

Raised by functions to indicate the argument is certainly outside the domain of the function.

B.2 Class Real

The real numbers interface is realized by the class *Real*.

B.2.1 Construction, destruction, assignment

The following constructors are available to the user:

```
Real::Real(const double src = 0);
```

Constructs a *Real* from a value in double precision, also acting as a default constructor for the value 0. The argument is taken to be exact, i.e. for example

the real number constructed by *Real*(0.1) is not the same as the real number 0.1, but rather to what the compiler thinks is its the closest *double* to 0.1.

Use for constants when you are certain they are correctly represented in double precision (e.g. integers up to 2^{53} are all correctly representable). If in doubt, use the string initialization form.

```
Real::Real(const char *src);
```

Constructs a *Real* from a decimal string. The string is taken to be exact. *Real*("0.1") will define the correct value, but *Real*("3.1415926") or even a 1000000-digit decimal representation will not define the number π (and neither would *Real*(*M_PI*)).

If a rational number is not correctly representable as a decimal, use division of real numbers to define it. For example, $\frac{1}{3}$ should be constructed as *Real*(1)/3[1].

```
typedef const char* (*OracleFunction) (unsigned precision);
Real::Real(OracleFunction oracle);
```

This constructor can be used to construct real numbers by an user-supplied function that can give decimal approximations of a real number for any precision (i.e. an "oracle" function).

This constructor is provided to give the system the possibility to work with numbers supplied from an external source (e.g. a human or a random number generator), which can possibly be non-computable. Other uses of the constructor are also possible[2], but those are covered by the more convenient and efficient real functions layer.

The oracle function is being called for increasing values of the *precision* parameter, which specifies that the function should try to present an approximation with this precision in 32-bit words, or roughly 10 decimal digits per unit of precision.

The function may choose to provide more or less accurate values, and the system takes them to be correct up to a unit in the last place (*ulp*) of the string the function returns. The requirement for the function is to represent *one* real number, i.e. to make sure that the intervals $[x - ulp, x + ulp]$ overlap for all values x that the function returns with *ulp* defined by the length of the decimal representation after the decimal point, and that the *ulp* values returned for different *precision* converge to zero, i.e. the for every k there is a value n such that for all *precision* $\geq n$, the length of the decimal representations returned by the oracle function is at least k.

The behavior of the system is not defined if the oracle function does not satisfy these requirements.

```
Real::Real(const Real &src);
```

Copy constructor, used to make a copy of a real value.

[1] It suffices to have one *Real* in the division to force division of real numbers. Division of a real number by an integer is faster and will be correct as long as the divisor can be represented as a 32-bit integer.

[2] e.g. defining real numbers by limitation

```
Real::~Real();
```
Destructor, called when a variable goes out of scope or a pointer is deleted.

```
Real& Real::operator = (const Real &rhs);
```
Assignment operator. Updates a real variable with a new value.

B.2.2 Operators

```
Real Real::operator - () const;
```
Negation.

```
Real operator + (const Real &lhs, const Real &rhs);
Real operator - (const Real &lhs, const Real &rhs);
Real operator * (const Real &lhs, const Real &rhs);
```
Addition, substraction and multiplication.

```
Real operator / (const Real &lhs, const Real &rhs);
```
Division. Not defined for *rhs* $== 0$.

```
Real operator * (const Real &lhs, int rhs);
Real operator * (int lhs, const Real &rhs);
Real operator / (const Real &lhs, int rhs);
Real operator / (int lhs, const Real &rhs);
```
Faster versions of multiplication and division by integer. The division by integer will cause a *DomainException* if *rhs* is zero, and the division of integer by real is not defined for *rhs* $== 0$.

```
Real::Real& operator += (const Real &rhs);
Real::Real& operator -= (const Real &rhs);
Real::Real& operator *= (const Real &rhs);
Real::Real& operator /= (const Real &rhs);

Real::Real& operator *= (int rhs);
Real::Real& operator /= (int rhs);
```
Updating versions of the operators. All of them are just shorthand forms for the operator followed by assignment.

B.2.3 Built-in constants and functions

```
extern const Real Pi;
```
The constant π.

```
extern const Real Ln2;
```
The constant $\ln 2$.

```
Real recip(const Real &arg);
```
Reciprocal, $\frac{1}{arg}$. Not defined for *arg* $== 0$.

```
Real abs(const Real &arg);
```
Absolute value, $|arg|$. The result is a non-negative real number.

`Real sqrt(const Real &arg);`

Square root, $\sqrt{arg}$. Not defined for negative arguments. The result is a non-negative real number.

`Real rsqrt(const Real &arg);`

Reciprocal square root, $\frac{1}{\sqrt{arg}}$. Not defined for 0 and negative arguments. The result is a positive real number.

`Real log(const Real &arg);`

Natural logarithm, $\ln arg$. Not defined for 0 and negative arguments.

`Real exp(const Real &arg);`

Exponent, e^{arg}. The result is a positive real number.

`Real sin(const Real &arg);`

Sine, $\sin arg$. The result is in the range $[-1, 1]$.

`Real cos(const Real &arg);`

Cosine, $\cos arg$. The result is in the range $[-1, 1]$.

`Real tan(const Real &arg);`

Tangent, $\tan arg = \frac{\sin arg}{\cos arg}$. Not defined for $arg == (2k + 1)\frac{\pi}{2}$ for an integer k.

`Real asin(const Real &arg);`

Arcsine, $\arcsin arg$. Defined only for $arg \in [-1, 1]$. The result is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

`Real acos(const Real &arg);`

Arccosine, $\arccos arg$. Defined only for $arg \in [-1, 1]$. The result is in the range $[0, \pi]$.

`Real atan(const Real &arg);`

Arctangent, $\arctan arg$. The result is in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$.

`Real atan2(const Real &y, const Real &x);`

Arctangent of $\frac{y}{x}$, using the signs of both arguments to compute the angle. Can be used to compute the argument of a complex number, or the angle between the origin of the plain and the point with coordinates (x, y).

The function is not defined[3] for the ray $y == 0, x <= 0$. The result is in the range $(-\pi, \pi)$.

[3]The mathematical function has a discontinuity at these points, which makes it non-computable. Excluding the points of discontinuity from the domain of the function makes it computable.

B.2.4 Comparison and truncation

`bool Real::`*IsNegative*`() const;`

Returns *true* if the number is negative and *false* if the number is positive. Not defined[4] if the number is zero.

`bool Real::`*IsPositive*`() const;`

Returns *false* if the number is negative and *true* if the number is positive. Not defined if the number is zero.

`bool Real::`*IsNonZero*`() const;`

Returns *true* if the argument is non-zero. Not defined if the number is zero. Can be used to cause the computation to be performed at a certain spot in the user's code or to make sure that numbers that are close to zero do not get printed as "probable zero".

```
bool operator < (const Real &lhs, const Real &rhs);
bool operator > (const Real &lhs, const Real &rhs);
bool operator != (const Real &lhs, const Real &rhs);
```

Comparison between two real numbers. Shorthand forms for substraction followed by resp. *IsNegative*, *IsPositive* and *IsNonZero*. Not defined if the two numbers are equal.

The equality test is undecidable, i.e. no function can exist that can always give a positive answer when two numbers are equal and not complete or give a negative answer if they are not. The system does not provide this test, nor the non-strict inequalities which, without the possibility to recognize equality, coincide with their strict counterparts.

`const Real& Real::`*ForceNonZero*`() const;`

Uses *IsNonZero* to force computation until the number can be separated from zero. Will loop indefinitely or cause an exception if the number is an actual zero. Used to make sure that numbers that are close to zero do not get printed as "probable zero".

B.2.5 Conversion to other types

`double Real::`*AsDouble*`() const;`

Conversion to *double*. Returns a double precision number which is at most 1 ulp away from the real number. The value returned is not always an IEEE-correct approximation to the number[5].

Numbers that are below the exponent range of *double* are converted to 0 and numbers that are above the range are mapped to ∞ with the appropriate sign.

[4] Again, the discontinuity in the discrete function is avoided by removing the points of discontinuity from its domain.

[5] The map between real values and their IEEE-correct representations is discontinuous thus non-computable. Our conversion is computable, but it is not a function in the sense of real analysis, because it can map different representations of the same real number into different *double* values.

```
char* Real::AsDecimal(char *buffer, unsigned len) const;
```

Creates a decimal approximation of the real number that fits in *len* characters, which is at most 1 ulp away from the number, and returns a pointer to *buffer*. The representation is not always correctly rounded[6], nor the first *len* digits of the number's infinite decimal representation[7].

Scientific notation is used if the number does not fit in the space provided. This also includes the cases where the number is smaller than the smallest number that can be written in fixed notation in the space provided (i.e. 10^{-len}). Because this can lead to infinite re-iteration if the number is 0, the system will return "probable zero" if the number is smaller than 10^{-2len} and is not distinguishable from zero at the current working precision. If this behavior is not desired (i.e. is the user knows the number is not zero), use *ForceNonZero* to make sure the computation is performed until number is separated from zero, for example

```
printf("%s", x.ForceNonZero().AsDecimal(buf, len));
```

The *buffer* must have enough space to accommodate *len* characters, and *len* must be at least 10.

B.2.6 Stream input and output

```
std::istream& operator >>(std::istream &in, Real &r);
```

Stream input. Reads a string from the input stream and creates a real number from it. The string can be of arbitrary (finite) length and is taken to be an exact decimal representation of the number.

```
std::ostream& operator <<(std::ostream &out, const Real& r);
```

Stream output. Sends a decimal approximation to *r* that is at most 1 ulp away to the output stream. The representation need not be correctly rounded nor the beginning of the number's infinite decimal representation[8].

If the number is smaller than 10^{-2prec} (for default and scientific notation), where *prec* is the output precision (set through *setprecision* below), and at the current working precision cannot be distinguished from 0, the system will print "probable zero" to avoid infinite loops if a real zero needs to be printed. If this behavior is not wanted, use *ForceNonZero* to force computation until separation from zero can be ensured, for example:

```
cout << x.ForceNonZero();
```

The output is influenced by the following stream manipulators:

setprecision(x) sets the precision in decimal digits. The digits of the exponent are not counted. The actual count of the digits depends on the notation

[6] for the same reasons as above

[7] e.g. the number 1.01000... can be converted to the decimal representation 1.0 as well as to 1.1, same reasons as above

[8] for the same reasons as in *AsDecimal* above

fixed sets fixed notation, the number is printed with the specified digits of precision after the decimal point, regardless how long the string needs to be to ensure that

scientific scientific (exponent) notation will be used, where the mantissa will have one non-zero decimal before the decimal point and the specified digits of precision after the decimal point

(notation not set) (default) prints in fixed notation if the number can fit in the specified digits of precision. If it cannot, prints in scientific notation. In both cases, the total number of digits printed (excluding the exponent) will be as specified

showpoint the trailing zeros and decimal point will be displayed

noshowpoint (default) the trailing zeros and decimal point will not be shown

showpos positive numbers will have a leading '+'

noshowpos (default) no leading '+' will be shown

setw(x) sets the field width (only applies to the next thing printed)

setfill(x) sets the character for filling the field

left the number will be left-justified in the field

right the number will be right-justified (default)

internal the sign will remain to the left, while the number will be flushed to the right

lowercase (default) use lowercase 'e' for the exponent

uppercase use uppercase 'E' for the exponent.

B.3 The functions interface: Class Estimate

This interface is meant to be used for the implementation of user-defined functions or user-defined real constants. To make the functions useable on the numbers level, they have to be defined according to one of the signatures specified in Section B.4.

On this level the functions are instantiated for different types which share the same interface. The interface is that of the class *Estimate*, which we are describing in this section.

The functions on this level work with approximations to the number. They return approximations to the result of the application of the function or operator. The nullary functions (constants or functions on integers) take an additional precision argument which indicates how precise the approximation should try to be. This argument is implicit in the functions that take real arguments

and can be recovered through *Estimate*::*GetPrecision*() on one of the real arguments. The precision indicates the approximation should try to be accurate to the specified number of 32-bit words of relative precision. The built-in functions try to achieve this[9].

The correctness requirement for the user functions is that they produce overlapping intervals, and that the error in the produced approximations as a function of the precision converges to zero, i.e. every function should satisfy the requirement that for every ε there are numbers n and δ, such that whenever the function is applied to arguments satisfying *GetPrecision*() $\geq n$ and *GetError*() $\leq \delta$, the returned interval satisfies *GetPrecision*() $\geq n$ and *GetError*() $\leq \varepsilon$. In order to achieve the best possible efficiency in the system, the user should try to produce approximations according to the precision specification.

From a user's point of view, functions that satisfy the correctness requirement (this includes the built-in functions), can be assumed to compute real numbers exactly. The errors they produce are handled automatically. The only errors that the user needs to address are the result of finitely approximating an infinite sequence. An upper bound for such an error should be explicitly specified via a call to *Estimate*::*AddError*.

Some of the operations on this level are called "weak" operations. This is to signify that they extract properties of the current approximations, which are not necessarily properties of the real number being approximated. Still, the weak operations can be useful to choose a control path when both control paths return the same end result, e.g. a *weak_lt* is used to switch between a control path that computes sin in the range $[0, \frac{\pi}{2}]$ and one that computes it for the range $[\frac{\pi}{2}, \pi]$. Both can compute the approximation even if the number is slightly outside that range, but use different algorithms and will yield results with different accuracy.

B.3.1 Conversion from other types

```
Estimate::Estimate(double v = 1.0);
Estimate::Estimate(const char *val);
```

Conversion from *double* or a decimal string. The argument is taken to be correct, and the resulting *Estimate* will be an approximation to that number (exact in the case of *double* and with decreasing error value as the precision increases in the case of a decimal string).

B.3.2 Error manipulation

```
Estimate Estimate::GetError() const;
Estimate& Estimate::SetError(const Estimate &err);
Estimate& Estimate::AddError(const Estimate &err);
```

Get, set and add to the error value in the approximation. The most often used function is *AddError*, which can be used to add the error resulting from imperfectly approximating a transcendental number via a finite part of an infinite sequence (see Section A.2 for example).

[9] but lose a few bits due to rounding errors of the basic operators

GetError returns an exact approximation (i.e. one having error 0), and the other functions use a value which is not smaller than the largest one within the argument interval.

Example: if *a* represents the real number interval $[0, 1]$, *a.GetError*() will return the interval $[0.5, 0.5]$, *a.SetError*(1) will make a representation of the interval $[-0.5, 1.5]$, and *a.AddError*(a) would return the interval $[-1, 2]$.

```
i32 Estimate::GetRelativeError() const;
```

Returns a lower bound for the number of digits in the mantissa of the approximation that are within 1-ulp of the real number. Used by the system in the process of obtaining approximations for conversion to *double* or decimal string.

```
u32 Estimate::GetPrecision() const;
Estimate& Estimate::SetPrecision(u32 prec);
```

Get and set the current working precision of the number in 32-bit words. This value controls how precise functions working on this argument should try to be (see Section A.2 for example).

B.3.3 Interval truncation

The truncation functions that follow are useful to compute functions that have domains with closed ends. They remove the specified parts of the approximating interval, leaving only the part that is a valid argument for the function. They raise an exception if the argument lies entirely in the unwanted part of the real line.

```
Estimate TruncateNegative(const char *origin = "Truncate") const;
```

Truncates the negative part of the interval. For example, $[-1, 2]$ will be truncated to $[0, 2]$, $[-2, -1]$ will raise a *DomainException*(*origin*), and $[1, 2]$ will remain unchanged.

```
Estimate TruncateBelow(double l,
   const char *origin = "Truncate") const;
Estimate TruncateBelow(const Estimate &l,
   const char *origin = "Truncate") const;
```

Truncates the parts below a certain lower limit. The error in *l* will appear in the result, e.g. if *a* is $[0, 2]$ and *l* is $[0.75, 1.25]$, *a.TruncateBelow*(*l*) will be $[0.75, 2.25]$. To make sure the limit is exact, use a *double* value for *l*.

```
Estimate TruncateAbove(double u,
   const char *origin = "Truncate") const;
Estimate TruncateAbove(const Estimate &u,
   const char *origin = "Truncate") const;
```

Truncates the parts above a certain upper limit. Use a *double* constant for *u* to avoid introducing extra error in the result.

```
Estimate TruncateTo(double l, double u,
   const char *origin = "Truncate") const;
```

```
Estimate TruncateTo(const Estimate &l, const Estimate &u,
   const char *origin = "Truncate") const;
```

Truncates the input interval to fit into the specified interval.

B.3.4 Operators

```
Estimate operator - (const Estimate &arg);

Estimate operator + (const Estimate &lhs, const Estimate &rhs);
Estimate operator - (const Estimate &lhs, const Estimate &rhs);
Estimate operator * (const Estimate &lhs, const Estimate &rhs);
Estimate operator / (const Estimate &lhs, const Estimate &rhs);

Estimate operator * (const Estimate &lhs, i32 rhs);
Estimate operator * (i32 lhs, const Estimate &rhs);
Estimate operator / (const Estimate &lhs, i32 rhs);
Estimate operator / (i32 lhs, const Estimate &rhs);
```

Negation, addition, substraction, multiplication and division, the last two also in a faster form with one integer argument.

Division needs the argument to be non-zero, thus it will raise a *PrecisionException* if the right-hand side is an interval that contains zero.

The division by integer form will raise a *DomainException* if the integer divisor is zero.

```
Estimate Estimate::operator << (i32 howmuch) const;
Estimate Estimate::operator >> (i32 howmuch) const;
```

Binary shift by *howmuch* bits, i.e. $a << n = a2^n$ and $a >> n = a2^{-n}$. Very fast multiplication by a power of two.

B.3.5 Built-in constants and functions

```
Estimate pi(unsigned int prec);
```

The constant π. Returns an approximation which has close to *prec* correct 32-bit words.

```
Estimate ln2(unsigned int prec);
```

The constant $\ln 2$.

```
Estimate recip(const Estimate &arg);
```

Reciprocal, $\frac{1}{arg}$. Raises a *PrecisionException* if the argument is an interval that contains 0.

```
Estimate abs(const Estimate &arg);
```

Absolute value, $|arg|$. The resulting interval may contain zero, but no negative real number.

```
Estimate sqrt(const Estimate &arg);
```

Square root, $\sqrt{arg}$. If the argument interval does not intersect the domain of the function, raises a *DomainException*. Otherwise, the interval is truncated

only to its valid part, i.e. the intersection of the domain of the function and the argument interval.

`Estimate rsqrt(const Estimate &arg);`

Reciprocal square root, $\frac{1}{\sqrt{arg}}$. Raises a *DomainException* for argument intervals that do not intersect the domain, and a *PrecisionException* for arguments that contain zero.

`Estimate log(const Estimate &arg);`

Natural logarithm, $\ln arg$. Raises a *DomainException* for argument intervals that do not intersect the domain, and a *PrecisionException* for arguments that contain zero.

`Estimate exp(const Estimate &arg);`

Exponent, e^{arg}.

`Estimate sin(const Estimate &arg);`

Sine, $\sin arg$. The resulting interval may contain numbers outside the range $[-1, 1]$.

`Estimate cos(const Estimate &arg);`

Cosine, $\cos arg$. The resulting interval may contain numbers outside the range $[-1, 1]$.

`Estimate tan(const Estimate &arg);`

Tangent, $\tan arg = \frac{\sin arg}{\cos arg}$. Raises a *PrecisionException* for arguments that contain $(2k + 1)\frac{\pi}{2}$ for an integer k.

`Estimate asin(const Estimate &arg);`

Arcsine, $\arcsin arg$. Raises a *DomainException* for intervals that do not intersect $[-1, 1]$ and truncates the argument interval to fit the domain.

`Estimate acos(const Estimate &arg);`

Arccosine, $\arccos arg$. Raises a *DomainException* for intervals that do not intersect $[-1, 1]$ and truncates the argument interval to fit the domain.

`Estimate atan(const Estimate &arg);`

Arctangent, $\arctan arg$.

`Estimate atan2(const Estimate &y, const Estimate &x);`

Arctangent of $\frac{y}{x}$, using the signs of both arguments to compute the angle. Can be used to compute the argument of a complex number, or the angle between the origin of the plain and the point with coordinates (x, y).

Raises a *PrecisionException* if the arguments contain points with $y == 0$ and $x <= 0$.

B.3.6 Strong comparisons

Strong comparisons return *true* if the comparison is true for any real number in the interval, i.e. if the approximated real number satisfies the inequality.

`bool Estimate::`*IsPositive*`() const;`
Returns *true* if *this* $\subseteq (0, +\infty)$, and *false* otherwise.

`bool Estimate::`*IsNegative*`() const;`
Returns *true* if *this* $\subseteq (-\infty, 0)$, and *false* otherwise.

`bool Estimate::`*IsNonZero*`() const;`
Returns *false* if $0 \in$ *this*, and *true* otherwise.

```
bool Estimate::operator < (const Estimate &rhs) const;
bool Estimate::operator > (const Estimate &rhs) const;
bool Estimate::operator != (const Estimate &rhs) const;
```

Comparison operators, shorthand forms for substraction followed by resp. *IsNegative*, *IsPositive*, *IsNonZero*.

If a value of *true* is returned, the real numbers satisfy the inequality, but a value of *false* means that either they do not satisfy it, or that this cannot be shown from the current approximation.

B.3.7 Weak discrete functions

Weak functions work with the current approximation, more specifically, with the center of the approximating interval. They ignore the error information and may give information that would be wrong for the approximated real number.

They are all discrete functions that would be non-computable on the real numbers level, but have a clearly defined meaning for its current approximation.

They are to be used to differentiate between control paths that compute the same thing via different algorithms (possibly with different accuracy), or for debugging or progress reports.

```
bool Estimate::weak_IsPositive() const;
bool Estimate::weak_IsNegative() const;
bool Estimate::weak_IsNonZero() const;

bool Estimate::weak_lt(const Estimate &rhs) const;
bool Estimate::weak_eq(const Estimate &rhs) const;
bool Estimate::weak_gt(const Estimate &rhs) const;

bool Estimate::weak_le(const Estimate &rhs) const;
bool Estimate::weak_ne(const Estimate &rhs) const;
bool Estimate::weak_ge(const Estimate &rhs) const;
```

Weak comparisons: positivity test, negativity test, non-zero[10], less-than, equal, greater-than, less-than-or-equal, not-equal, greater-than-or-equal.

[10]Some implementations define this to be always *true*, because they do not allow a zero to be the center of an approximation.

`Estimate Estimate::weak_round() const;`

Rounding. Returns an exact *Estimate*, which is the integer closest to the center of the interval. Can be used to compute the value of periodic functions.

`i32 Estimate::weak_normalize() const;`

Normalization: returns an integer such that $a >> a.Normalize()$ is within $[0.5, 1)$. Used to compute logarithms.

`double Estimate::weak_AsDouble() const;`

Returns a *double* approximation to the center of the interval, at most $\frac{1}{2}$ ulp away from it, correctly rounded according to the IEEE-754 specification[11].

`char *Estimate::weak_AsDecimal(char *buffer, u32 buflen) const;`

Returns a decimal representation of the center of the interval which fits in *buflen* characters and is at most $\frac{1}{2}$ ulp away. Fixed notation is normally used, switched to scientific if the number cannot fit in the space provided (*buflen* has to be 10 characters minimum).

`std::ostream& operator <<(std::ostream &os, const Estimate &e);`

Stream output, prints a number which is at most $\frac{1}{2}$ ulp away from the center of the interval. Influenced by the same set of format manipulators as the stream output for real numbers.

B.4 Macros linking the functions and numbers interfaces

In order to use functions defined on the functions layer on *Real* arguments, the user must create a mapping of the function using one of the linking macros.

There are linking macros for the following types of functions:

- nullary functions, i.e. real constants
- nullary functions with int, i.e. functions taking one integer argument and returning a real number
- unary real functions
- unary functions with int, i.e. functions taking one real and one integer arguments
- binary real functions
- binary functions with int, i.e. functions taking two real and one integer arguments
- real functions on arrays
- real functions on arrays with an additional integer argument

[11] This specification is not correctly implemented in the current version.

The user's functions have to be function templates parameterized by the type of the approximation object. The approximation objects for which they will be instantiated all share the interface of *Estimate*, described in Section B.3.

B.4.1 Nullary functions (constants)

Defined using this form:

```
template <class TYPE>
TYPE name(unsigned int prec);
```

The macro *CreateNullaryRealFunction*(*name*) maps such a function to the following real function:

```
Real name ();
```

Alternatively, the macro *CreateRealConstant*(*const_name*, *fun_name*) maps the function *fun_name* into the constant *const_name* defined as

```
const Real const_name;
```

For example, *CreateRealConstant*(*Pi*, *pi*) is used in the code of the system to define the built-in constant *Pi* from the function *pi*.

B.4.2 Nullary functions with integer argument

Defined using this form:

```
template <class TYPE>
TYPE name(unsigned int prec, UserInt uint);
```

The macro *CreateIntRealFunction*(*name*) maps such a function to the following real function:

```
Real name (UserInt uint);
```

B.4.3 Unary functions

Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE &arg);
```

The macro *CreateUnaryRealFunction*(*name*) maps such a function to the following real function:

```
Real name (const Real& arg);
```

For example, *CreateUnaryRealFunction*(*sin*) is used in the source code of the system to define the real number function *sin* from the function-layer *sin*.

B.4.4 Unary functions with integer argument

Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE& arg, UserInt uint);
```

The macro *CreateUnaryAndIntRealFunction*(*name*) maps such a function to the following real function:

```
Real name (const Real& arg, UserInt uint);
```

B.4.5 Binary functions

Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE &lhs, const TYPE &rhs);
```

The macro *CreateBinaryRealFunction*(*name*) maps such a function to the following real function:

```
Real name (const Real& lhs, const Real& rhs);
```

For example, *CreateBinaryRealFunction*(*atan2*) is used in the source of the system to define the real number function *atan2* from the function-layer *atan2*.

B.4.6 Binary functions with integer argument

Defined using this form:

```
template <class TYPE>
TYPE name(const TYPE& lhs, const TYPE &rhs, UserInt uint);
```

The macro *CreateBinaryAndIntRealFunction*(*name*) maps such a function to the following real function:

```
Real name (const Real& lhs, const Real& rhs, UserInt uint);
```

B.4.7 Array functions

Defined using this form:

```
template <class TYPE, class ARRAY>
TYPE name(ARRAY &arg);
```

where *ARRAY* is a type with the following interface:

```
template <class TYPE>
class ArrayInterface {
public:
    long size();
    TYPE& operator[] (long index);
};
```

(indexed array elements can be retrieved or updated; no pointer or prev/next operations are supported and no bounds checking is performed)

The macro *CreateArrayRealFunction*(*name*) maps such a function into the triple:

```
void name(Real *ptr, long count);
void name(std::valarray<Real> &arr);
void name(std::vector<Real> &arr);
```

For an example of the use of this, see `examples/linear.cpp`.

B.4.8 Array functions with integer argument

Defined using this form:

```
template <class TYPE>
TYPE name(ARRAY &arg, UserInt int);
```

where *ARRAY* is as above.

The macro *CreateArrayAndIntRealFunction*(*name*) maps such a function into the triple:

```
void name(Real *ptr, long count, UserInt uint);
void name(std::valarray<Real> &arr, UserInt uint);
void name(std::vector<Real> &arr, UserInt uint);
```

Bibliography

[1] Baillon, J, Bruck, R.E., *The rate of asymptotic regularity is* $0(\frac{1}{\sqrt{n}})$. Theory and applications of nonlinear operators of accretive and monotone type, Lecture Notes in Pure and Appl. Math. **178**, pp. 51-81, Dekker, New York (1996).

[2] Bauer, A., *Realizability as the Connection between Computable and Constructive Mathematics*. Draft, available at `http://math.andrej.com/category/papers/`.

[3] Berger, U., Oliva, P., *Modified Bar Recursion and Classical Dependent Choice.* Lecture Notes in Logic **20**, Baaz, M., Friedman, S., Krajícek, J. (eds.), Logic Colloquium '01, Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic, Vienna, August 6 - 11, 2001, pp. 89–107 (2005).

[4] Berinde, V., *Iterative approximation of fixed points*. Efemeride, Baia Mare, xii+283pp. (2002).

[5] Blum, L., Shub, M., Smale, S., *On the theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*. Bulletin of the Amer. Math. Soc., **21**(1), pp. 1–46 (1989).

[6] Brattka, V., Hertling, P., *Feasible Real Random Access Machines*. Journal of Complexity **14**, Issue 4, pp. 490–526 (1998).

[7] Briggs, K., *Implementing exact real arithmetic in python, C++ and C*, to appear in Journal of theoretical computer science
see also `http://more.btexact.com/people/briggsk2/xrc.html`

[8] Browder, F.E., *Nonexpansive nonlinear operators in a Banach space*. Proc. Nat. Acad. Sci. U.S.A. **54**, pp. 1041-1044 (1965).

[9] Browder, F.E., Petryshyn, W.V., *The solution by iteration of nonlinear functional equations in Banach spaces*. Bull. Amer. Math. Soc. **72**, pp. 571-575 (1966).

[10] Cheney, E. W. *Introduction to Approximation Theory*, 2nd ed. Providence, RI: Amer. Math. Soc. (1999).

[11] Clarkson, J.A., *Uniformly convex spaces*. Trans. Amer. Math. Soc. **40**, pp. 396-414 (1936).

[12] O'Connor, R., *Few Digits*,
available at `http://r6.ca/FewDigits/`

[13] Cook, S.A., Kapron, B.M., *Characterizations of the basic feasible functionals of finite type.* Feasible Mathematics: A Mathematical Sciences Institute Workshop, Birkhauser, Eds. S. Buss, P. Scott, pp. 71–96 (1990).

[14] Cook, S., Urquhart, A., *Functional interpretations of feasibly constructive arithmetic.* Annals of Pure Applied Logic **63**, pp. 103–200 (1993).

[15] Coquand, T., Hofmann, M., *A new method for establishing conservativity of classical systems over their intuitionistic version*. Lambda-calculus and logic. Mathematical Structures in Computer Science **9**, pp. 323–333 (1999).

[16] Diaz, J.B., Metcalf, F.T., *On the structure of the set of subsequential limit points of successive approximations*. Bull. Amer. Math. Soc. **73**, pp. 516-519 (1967).

[17] Diaz, J.B., Metcalf, F.T., *On the set of subsequential limit points of successive approximations*. Trans. Amer. Math. Soc. **135**, pp. 459-485 (1969).

[18] Dotson, W.G., Jr., *On the Mann iterative process*. Trans. Amer. Math. Soc. **149**, pp. 65-73 (1970).

[19] Edalat, Abbas; Sünderhauf, Philipp *A domain-theoretic approach to computability on the real line.* Theoret. Comput. Sci. 210, no. **1**, pp. 73–98 (Reviewer: Klaus Weihrauch) (1999).

[20] Edalat, A., *Exact Real Number Computation Using Linear Fractional Transformations.* Final Report on EPSRC grant GR/L43077/01
Available at `http://www.doc.ic.ac.uk/~ae/exact-computation/exactarithmeticfinal.ps.gz`

[21] Ershov, A.G., Kashevarova, T.P., *Interval Mathematical Library Based on Chebyshev and Taylor Series Expansion*. Reliable Computing Vol. **11**, No. 5 (2005).

[22] Escardo, M., *PCF extended with real numbers*. Theoretical Computer Science, Elsevier, volume 162, issue 1, pp. 79-115 (1996).

[23] Friedman, H., *Classical and intuitionistically provably recursive functions*. Müller, G.H., Scott, D.S. (eds.), Higher Set Theory, pp. 21–27. Springer LNM **669** (1978).

[24] Garcia-Falset, J., Llorens-Fuster, E., Prus, S., *The fixed point property for mappings admitting a center*. Talk given at '7th International Conference on Fixed Point Theory and its Applications', July 17-23, 2005, Guanajuato, Mexico.

[25] Gerhardy, P., Kohlenbach, U., *General logical metatheorems for functional analysis*. Sumitted.
available at `http://www.mathematik.tu-darmstadt.de/~kohlenbach/genmeta.ps.gz`

[26] Goebel, K., Kirk, W.A., *A fixed point theorem for asymptotically nonexpansive mappings*. Proc. Amer. Math. Soc. **35**, pp. 171-174 (1972).

[27] Goebel, K., *Concise Course of Fixed Point Theorems*. Yokohama Publishers, Yokohama, iii+182pp. (2002).

[28] Gödel, K., *Zur intuitionistischen Arithmetik und Zahlentheorie*. Ergebnisse eines Matematischen Kolloquiums, vol. **4**, pp. 280–287 (1933).

[29] Gödel, K., *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes*. Dialectica **12**, pp. 280–287 (1958).

[30] Göhde, D., *Zum Prinzip der kontraktiven Abbildung*. Math. Nachrichten **30**, pp. 251-258 (1965).

[31] Green, R., *Faster Math Functions*. Game Developers Conference (2002).
available at `http://www.research.scea.com/research/pdfs/RGREENfastermath_GDC02.pdf`

[32] Groetsch, C.W., *A note on segmenting Mann iterates*. J. of Math. Anal. and Appl. **40**, pp. 369-372 (1972).

[33] Grzegorczyk, A., *Some classes of recursive functions*. Rozprawny Matematyczne, 46 pp., Warsaw (1953).

[34] Grzegorczyk, A., *On the definitions of computable real continuous functions*. Fundamenta Matematicae **44**, pp. 61–67 (1957).

[35] von Gudenberg, J.W., *Interval Arithmetic on Multimedia Architectures*. Reliable Computing Vol. **8** No. 4 (2002).

[36] Hanner, O., *On the uniform convexity of L_p and l_p*. Ark. Mat. **3**, pp. 239-244 (1956).

[37] Hida, Y., Li, X. S. and Bailey, D. H. , *Algorithms for Quad-Double Precision Floating Point Arithmetic*, 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society, pg. 155-162 (2001).

[38] Hillam, B., *A Generalization of Krasnoselski's Theorem on the Real Line*, Math. Magazine **47-48** pp. 167–168 (1974-1975).

[39] Hinman, P.G., *Recursion-theoretic hierarchies*. Perspectives in Mathematical Logic, Springer, xii+480 pp. (1978).

[40] Hofschuster, W., Krämer, W., Lerch, M., Tischler G., von Gudenberg, J.W., *The Interval Library fi_lib++ 2.0 Design, Features and Sample Programs*. Preprint 2001/4,Universität Wuppertal, (2001).
available at `http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_4.pdf`

[41] Howard, W.A., *Hereditarily majorizable functionals of finite type.* In: Troelstra (ed.), Metamathematical investigation of intuitionistic arithmetic and analysis, pp. 454-461. Springer LNM **344** (1973).

[42] Ishikawa, S., *Fixed points and iteration of a nonexpansive mapping in a Banach space*, Proc. Amer. Math. Soc. **59**, pp. 65-71 (1976).

[43] Jørgensen, K.F., *Finite Type Arithmetic: computable existence analysed by modified realizability and functional interpretation.* Master Thesis, University of Roskilde, viii+121 pages (2001).

[44] Kahan, W., *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating Point Arithmetic.*
available at `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.pdf`

[45] Kapron, B. M.; Cook, S. A. *A new characterization of type-2 feasibility.* SIAM J. Comput. **25**, no. 1, pp. 117–132 (1996).

[46] Khamsi, M.A. , Kirk, W.A., *An introduction to metric spaces and fixed point theory*, Wiley (2001).

[47] Kirk, W.A., *A fixed point theorem for mappings which do not increase distances.* Amer. Math. Monthly **72**, pp. 1004-1006 (1965).

[48] Kirk, W.A., Martinez-Yanez, C., *Approximate fixed points for nonexpansive mappings in uniformly convex spaces.* Annales Polonici Mathematici **51**, pp. 189-193 (1990).

[49] Kleene, S.C., *Recursive functions and intuitionistic mathematics*, Proceedings of the International Congress of Mathematicians, Cambridge, Mass., 1950, pp. 679–685. Americal Mathematical Society, Providence, R.I. (1952).

[50] Kleene, S.C., *Recursive Functionals and Quantifiers of Finite Types I.* Trans. Amer. Math. Soc. **91**, pp. 1–52 (1959).

[51] Kleene, S.C., *Countable Functionals.* In: A. Heyting (ed), Constructivity in Mathematics, North-Holland, Amsterdam, 81–100 (1959).

[52] Knuth, D., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Third Edition. Reading, Massachusetts: Addison-Wesley, xiv+762pp. (1997).

[53] Ko, K.-I., *Complexity Theory of Real Functions*, Birkhäuser, Boston-Basel-Berlin, x+309 pp. (1991).

[54] Kohlenbach, U., *Theory of majorizable and continuous functionals and their use for the extraction of bounds from non-constructive proofs: effective moduli of uniqueness for best approximations from ineffective proofs of uniqueness*(german). PhD Dissertation, Frankfurt (1990).

[55] Kohlenbach, U., *Effective bounds from ineffective proofs in analysis: an application of functional interpretation and majorization.* Journal of Symbolic Logic **57**, pp. 1239–1273 (1992).

[56] Kohlenbach, U., *Effective moduli from ineffective uniqueness proofs. An unwinding of de La Vallée Poussin's proof for Chebycheff approximation.* Annals of Pure and Applied Logic **64** pp. 27–94 (1993).

[57] Kohlenbach, U., *Mathematically strong subsystems of analysis with low rate of growth of provably recursive functionals.* Archive for Mathematical Logic **36**, pp. 31–71 (1996).

[58] Kohlenbach, U., *Analysing proofs in analysis.* W.Hodges, M. Hyland, C. Steinhorn, J. Truss, editors, Logic: from Foundations to Applications. European Logic Colloquium (Keele 1993), pp. 225–260, Oxford University Press (1996)

[59] Kohlenbach, U., *Elimination of Skolem functions for monotone formulas in analysis.* Archives for Mathematical Logic **37**, pp. 363–390 (1998).

[60] Kohlenbach, U., *On the no-counterexample interpretation.* Journal of Symbolic Logic **64**, pp. 1491–1511 (1999).

[61] Kohlenbach, U., *A quantitative version of a theorem due to Borwein-Reich-Shafrir.* Numer. Funct. Anal. and Optimiz. **22**, pp. 641–656 (2001).

[62] Kohlenbach, U., *On the computational content of the Krasnoselski and Ishikawa fixed point theorems.* Proceedings of the Fourth Workshop on Computability and Complexity in Analysis, J. Blanck, V. Brattka, P. Hertling (eds.), Springer LNCS 2064, pp. 119-145 (2001).

[63] Kohlenbach, U., *Uniform asymptotic regularity for Mann iterates.* J. Math. Anal. Appl. **279**, pp. 531-544 (2003).

[64] Kohlenbach, U., Leuştean, L., *Mann iterates of directionally nonexpansive mappings in hyperbolic spaces.* Abstr. Appl. Anal. **2003**, no.8, pp. 449-477 (2003).

[65] Kohlenbach, U., Oliva, P., *Proof mining: a systematic way of analysing proofs in analysis.* Proc. Steklov Inst. Math. **242**, 136-164 (2003).

[66] Kohlenbach, U., Lambov, B., *Bounds on iterations of asymptotically quasi-nonexpansive mappings.* Proceedings of the international conference on Fixed Point Theory and Applications, Valencia 2003, Falset, J.G., Fuster, E. L., Sims, B. (eds.), Yokohama Publishers, pp. 143-172 (2004).

[67] Kohlenbach, U., *Some logical metatheorems with applications in functional analysis.* Trans. Amer. Math. Soc. vol. 357, no. **1**, pp. 89-128 (2005).

[68] Kohlenbach, U., *Some computational aspects of metric fixed-point theory.* Nonlinear Analysis **61**, pp. 823–837 (2005).

[69] Kohlenbach, U., *Proof Interpretations and the Computational Content of Proofs*, draft.
Available at `http://www.mathematik.tu-darmstadt.de/~kohlenbach/newcourse.ps.gz`.

[70] Kolla, R., Vodopivec, A., von Gudenberg, J.W., *The IAX Architecture – Interval Arithmetic Extension*. Universität Würzburg, Institut für Informatik, Techn. Report TR225, April 1999.
available at `http://www2.informatik.uni-wuerzburg.de/mitarbeiter/wvg/Public/iax.ps.gz`

[71] Krasnoselski, M. A., *Two remarks on the method of successive approximation*. Usp. Math. Nauk (N.S.) **10**, pp. 123-127 (1955) (Russian).

[72] Kreisel, G., *On the interpretation of non-finitist proofs, part I*. Journal of Symbolic Logic **16**, pp. 241–267 (1951).

[73] Kreisel, G., *On the interpretation of non-finitist proofs, part II: Interpretation of number theory, applications*. Journal of Symbolic Logic **17**, pp. 43–58 (1952).

[74] Kreisel, G., *Interpretation of analysis by means of constructive functionals of finite types*. In: Constructivity in Mathematics, Heyting, A. (ed.). North-Holland (Amsterdam), pp. 101–128 (1959).

[75] Lambov, B., *Complexity and Intensionality in a Type-1 Framework for Computable Analysis*. Ong, L. (ed.), Computer Science Logic: 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005. Proceedings. Lecture Notes in Computer Science **3634**, pp. 442–461 (2005).

[76] Lambov, B., *Rates of convergence of recursively defined sequences*. Proceedings of the 6th Workshop on Computability and Complexity in Analysis (CCA 2004), Electronic Notes in Theoretical Computer Science, Volume 120, pp. 125-133 (2005).

[77] Lambov, B., *"RealLib, an Efficient Implementation of Exact Real Arithmetic"*, CCA 2005 - Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan. Informatik Berichte 326-7/2005 FernUniversität Hagen, Germany (2005).

[78] Lambov, B., *RealLib3 Manual*,
available at `http://www.brics.dk/~barnie/RealLib/`

[79] Machado, H.V., *A characterization of convex subsets of normed spaces*. Kodai Math. Sem. Rep. **25**, pp. 307-320 (1973).

[80] Mann, W.R., *Mean value methods in iteration*. Proc. Amer. Math. Soc. **4**, pp. 506-510 (1953).

[81] Matiyasevich, Y. V., *A sufficient condition for the convergence of monotone sequences*. Zapiski Nauchnykh Seminarov Leningradskovo Otdeleniya Matematicheskogo Instituta imeni V.A. Steklova. Akademii Nauk SSSR, Vol. **20** (1971), pp. 97–103. English translation in J. Sov. Math. 1, No. **1**, pp. 59–63 (1973).

[82] Mazur, S., *Computable Analysis*. Razprawy Matematyczne, **33**, Warsaw (1963).

[83] Melhorn, K., *Polynomial and abstract subrecursive classes*, Journal of Computer System Science **12** , pp. 147–178 (1976).

[84] Mehlhorn, K. and Schirra, S., *Generalized and improved constructive separation bound for real algebraic expressions*, Research Report, Max-Planck-Institut für Informatik, November (2000).

[85] Mosses, P. D., *Action Semantics*. Cambridge Tracts in Theoretical Computer Science **26**, Cambridge University Press (1992).

[86] Müller, N., *The iRRAM: Exact arithmetic in C++*. Computability and complexity in analysis. (Swansea, 2000). Lecture Notes in Computer Science **2064**. Springer (2001).
see also `http://www.informatik.uni-trier.de/iRRAM/`

[87] Naimpally, S.A., Singh, K.L., Whitfield, J.H.M., *Fixed points and sequences of iterates in locally convex spaces*. Topological methods in nonlinear functional analysis. Contemp. Math. **21**, Amer. Math. Soc., pp. 159-166 (1983).

[88] Odifreddi, P., *Classical Recursion Theory*. North-Holland, xvii+668 pp. (1989).

[89] Odifreddi, P., *Classical Recursion Theory, Volume II*. Studies in Logic and the Foundations of Mathematics **143**, xvi+949 pp. (1999).

[90] Opial, Z., *Weak convergence of the sequence of successive approximations for nonexpansive mappings*. Bull. Amer. Math. Soc. **73**, pp. 595-597 (1967).

[91] Orevkov, V.P., *A constructive mapping of the square onto itself displacing every constructive point*, Soviet Math. Doklady **4**, pp. 1253–1256 (1963).

[92] Parsons, C., *Proof theoretic analysis of restricted induction schemata*. Journal of Symbolic Logic vol. **36**, p. 361 (1971).

[93] Pour-El, M.B., Richards, J.I., *Computability in Analysis and Physics*. Springer, x+206 pp. (1989).

[94] Qihou, L., *Iteration sequences for asymptotically quasi-nonexpansive mappings.* J. Math. Anal. Appl. **259**, pp. 1–7 (2001).

[95] Qihou, L., *Iteration sequences for asymptotically quasi-nonexpansive mappings with error member*. J. Math. Anal. Appl. **259**, pp. 18-24 (2001).

[96] Qihou, L., *Iteration sequences for asymptotically quasi-nonexpansive mapping with an error member of uniform convex Banach space.* J. Math. Anal. Appl. **266**, pp. 468–471 (2002).

[97] Rhoades, B.E., *Fixed point iterations for certain nonlinear mappings*. J. Math. Anal. Appl. **183**, pp. 118-120 (1994).

[98] Ritchie, R.W., *Classes of recursive functions based on Ackermann's function*. Pacific Journal of Mathematics **15**, pp. 1027–1044 (1965).

[99] Schu, J., *Iterative construction of fixed points of strictly quasicontractive mappings*. Appl. Anal. **40**, pp. 67-72 (1991).

[100] Schu, J., *Weak and strong convergence to fixed points of asymptotically nonexpansive mappings*. Bull. Austral. Math. Soc. **43**, pp. 153-159 (1991).

[101] Schu, J., *Iterative construction of fixed points of asymptotically nonexpansive mappings*. J. Math. Anal. Appl. **158**, pp. 407–413 (1991).

[102] Schwichtenberg, H., *Constructive Analysis with Witnesses* (Marktoberdorf '03)
Available at `http://www.mathematik.uni-muenchen.de/~schwicht/papers/mod03/modart03.ps`

[103] Shioji, N., Tanaka, K., *Fixed point theory in weak second- order arithmetic*, Annals of Pure and Applied Logic **47**, 167-188 (1990).

[104] Simpson, S.G., *Subsystems of Second Order Arithmetic*. Perspectives in Mathematical Logic. Springer-Verlag, xiv+445 pp. (1999).

[105] Skordev, D., *Characterization of the computable real numbers by means of primitive recursive functions.* Computability and complexity in analysis (Swansea, 2000), Lecture Notes in Computer Science **2064**, pp. 296–309, Springer (2001).

[106] Specker, E., *Nicht konstruktiv beweisbare Sätze der Analysis*. The Jornal of Symbolic Logic, **14**(3), pp. 145–158 (1949).

[107] Srivastava, P., Srivastava, S.C., *On asymptotically quasinonexpansive families of mappings*. In: Nonlinear analysis and applications (St. Johns, 1981), Lecture Notes in Pure and Appl. Math. **80**, pp. 265-270, Dekker (1982).

[108] Tait, W.W., *Gödel's reformulation of Gentzen's First Consistency proof for arithmetic: the no-counterexample interpretation*. Bulletin of Symbolic Logic **11**, pp. 225–238 (2005).

[109] Tan, K.-K., Xu, H.K., *Fixed point iteration processes for asymptotically nonexpansive mappings*. Proc. Amer. Math. Soc. **122**, pp. 733-739 (1994).

[110] Troelstra, A.S. (editor), *Metamathematical investigation of intuitionistic arithmetic and analysis*. Springer LNM **344**, xii+485 pp. (1973).

[111] Turing, A., *On computable numbers, with an application to the "Entscheidungsproblem"*. Proceedings of the London Mathematical Society, **42**(2), pp. 230–265 (1936).

[112] Turing, A., *On computable numbers, with an application to the "Entscheidungsproblem". A correction.* Proceedings of the London Mathematical Society, **43**(2), pp. 544–546 (1937).

[113] Weihrauch, K., *Computable Analysis.* Springer, Berlin, x+285 pp. (2000).

[114] Xu,Y., *Ishikawa and Mann iterative processes with errors for nonlinear strongly accretive operator equations*. J. Math. Anal. Appl. **224**, pp. 91–101 (1998).

[115] *Boost Interval Arithmetic Library*.
available at `http://www.boost.org/libs/numeric/interval/doc/interval.htm`

[116] *The Gnu Multiple Precision arithmetic library*, `http://www.swox.com/gmp/`.

[117] IEEE Standards Committee 754, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York (1985). Reprinted in SIGPLAN Notices, 22(2):9–25 (1987).

[118] Intel Corp., *IA-32 Intel® Architecture Software Developer's Manual, Volumes 1-3*.
available at `http://developer.intel.com/design/pentium4/manuals/index_new.htm`

[119] Intel Corp., *Using SSE3 Technology in Algorithms with Complex Arithmetic*.
available at `http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/66717.htm`

[120] Intel Corp., *Next Generation Intel Processor: Software Developers Guide*.
available at `http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/66756.htm`