Towards a Complexity Theory for Approximation *

Karl Aberer † and Bruno Codenotti ‡

TR-92-012

February 1992

Abstract

This paper presents a novel approach to the analysis of numerical problems, which is closely related to the actual nature of numerical algorithms. In fact, models of computation are introduced which take into account such issues as adaptivity and error. Moreover, complexity vs error bounds and examples regarding the role of adaptivity are provided. Finally, it is shown that the overall approach fits naturally into an algebraic framework.

*The work of Bruno Codenotti was partially supported by the Italian National Research Council under the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottoprogetto 2". Bruno Codenotti is on leave from IEI-CNR, Pisa (ITALY).

†email: aberer@icsi.berkeley.edu

‡email: brunoc@icsi.berkeley.edu

1 Introduction

There is a lack of understanding among different scientific communities working on numerical problems from different viewpoints. The main conflict, reported for example in [Smale, 1990], is between numerical analysis and computer science. Numerical analysis is essentially concerned with computing with fixed-precision numbers intended to be approximations of real numbers. Computer science deals with discrete problems, so that the notion of numerical error is missing: data and computations are exact and researchers are devoted to find "good" or, possibly, "optimal" algorithms, with respect to given measures.

In this paper we propose a framework to analyze numerical problem, which is close to the practice of numerical computations, but is, at the same time, very simple, in order to allow an investigation by mathematical tools.

We first introduce some models of computation oriented towards the analysis of problems defined over fields of characteristic 0 (Section 2). A model of computation for this kind of problems has at least to take into account the following features:

1. *To separate the part of a numerical computation that uses the finite representation of real numbers and the arithmetic operations on them (which thus introduces errors), from the purely combinatorial part*, e.g., searching for a data with a given property or comparing two data. We achieve this goal by organizing a numerical computation in alternating combinatorial and arithmetic stages.

2. *To compare direct versus iterative methods.* Although these two types of methods are of very different nature, once one applies them to finite approximations of real numbers, then they become easily comparable. The tool we use to make this comparison is recursion, which we put as part of our models.

3. *To deal with errors arising in the computations* (i.e., truncation, roundoff and perturbation error). We achieve this goal by keeping additional information together with numerical data (i.e. an evaluation of the updated accuracy).

4. *To use the notion of problem conditioning* (e.g. the condition number of a matrix) *also as a complexity measure.* We do this by letting the "problem conditioning" play the same role, within the model, as the problem size.

5. *To compare adaptive and oblivious numerical algorithms.* We are able to perform this comparison by checking how conditional operations are used in the computations.

Analyzing numerical problems within the discrete framework peculiar to computer science, which can be done by assuming to perform basic computations on real numbers with unitary cost and no error, leads, e.g., to characterizations of complexity classes for numerical problems. Many results originally developed for problems defined over fields of characteristic 0, e.g. Csanky's algorithm for parallel matrix inversion [Csanky, 1976], have been extended to handle the case of finite fields. In this way, the algebraic structure of the problems has been

stressed, while the important issue of approximation over infinite fields has been left as an area of investigation for numerical analysts. A drawback of this situation is the following: when one tries to turn algorithms developed for the algebraic complexity framework into concrete algorithms, e.g., to be run on finite machines, then many computational obstacles come into play (see, for example, [Codenotti *et al.*, 1991]).

We face this problem by proposing a framework in which both the numerical analysis and computation theory approaches would fit, and show a number of results strongly relating complexity and accuracy (Section 3).

Many authors have tried to capture the many features of numerical problems, using different models and theories. Some were oriented toward quantifying the cost of computing up to a given accuracy [Traub *et al.*, 1988]; some others were directed to extending combinatorial notions (e.g., NP-completeness) to problems defined on continuous structures [Blum *et al.*, 1989]. Other theories have led to an information-theoretic view of computations [Chaitin, 1987]. Finally, several trials were oriented to evaluate the bit-complexity of problems, i.e. the number of bit-operations necessary to compute up to a given accuracy (see, e.g., [Brent, 1976]).

From the result in [Cai, Hartmanis, 1989], where it is proved that the Kolmogorov complexity of the real line is a fractal, the following observations naturally follow: (i) from a computational viewpoint, the representation of real numbers is a very intriguing issue; (ii) real numbers are computationally different.

These properties deeply reflect in this paper, where we find evidence of the non straightforward nature of approximating and computing with approximations.

On a different side, in [Smale, 1990], several questions of fundamental importance are addressed. In particular, we would like to underline the following:

- There exists a conflict between scientific computing and computer science;
- Foundations of numerical analysis are missing;
- There is not a formal definition of algorithm in Numerical Analysis;
- There exists "the problem of reconciling the digital machine with the continuous mathematics of scientific computation".

For what concerns the latter point, we believe that the situation is going to change, for at least two reasons: (i) the employment of probabilistic techniques in algorithmics has made necessary to extensively use continuous mathematics in computer science; (ii) the average case behaviour of algorithms can be analyzed only by using continuous mathematics.

Smale poses some other questions and, in particular, makes the observation that "complexity lower bounds of various types have a big future in numerical analysis".

To this extent, we show here a number of results, which are summarized in the following.

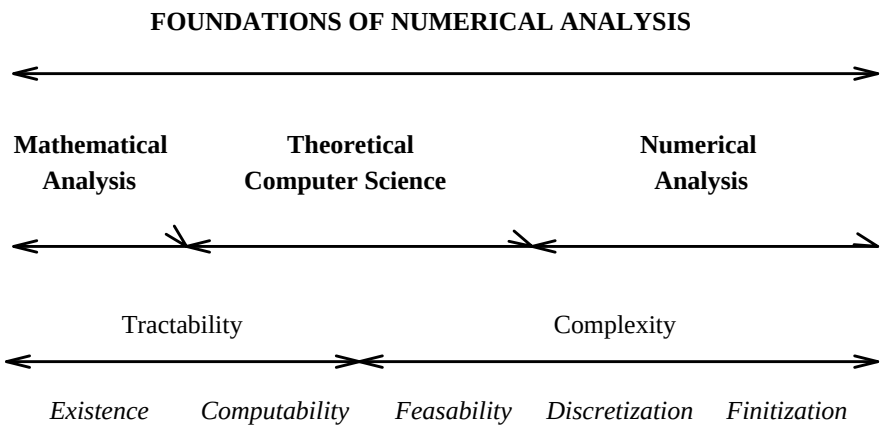

Figure 1: Foundations of numerical analysis

(1) The distinction between oblivious and adaptive algorithms is crucial. (see Section 3).

(2) For oblivious algorithms, we are able to detect an unavoidable extra error term, which we call representation error, and which can be added to the perturbation error to get a stronger lower bound on the error. (see Sections 3 and 4).

(3) We show under which conditions the representation error is lower bounded by the minimum depth or width of all computation graphs representing computations of f_n.

(4) We are able to relate error with parallel or sequential time, thus proving several complexity-error relationships.

(5) By using the previous arguments combined with arithmetic requirements, we prove complexity lower bounds for numerical computations by oblivious algorithms (see Sections 3 and 4).

Another goal of this paper is to put a bridge between the question "how fast can one compute?", typical of complexity theory, and the question "how accurate can one compute?", typical of numerical analysis. More precisely, we want to study the efficiency of numerical algorithms, by facing the two questions, "How accurate can fast algorithms be?" and "How fast can accurate algorithms be?". Our opinion is that quantitative answers to these last questions would give a new insight into the design and analysis of numerical computations. These issues are analyzed in Section 4, mainly by proposing several interesting case studies.

The investigation of Section 4 takes some motivations from [Smale, 1990], and has strong connections to [Traub *et al.*, 1988], [Blum *et al.*, 1989] and [Codenotti *et al.*, 1991].

The above discussion puts also into evidence that there is a need for a suitable language for computing with approximation. This issue is the topic of Section 5.

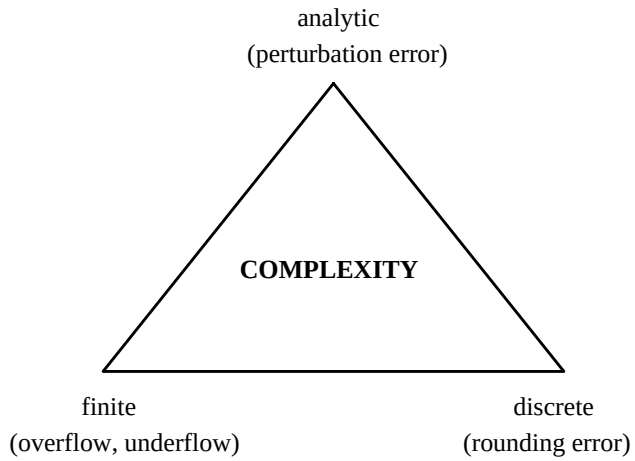

Figure 2: The complexity of a numerical problem is related to three sources of error.

Summarizing, the rest of this paper is organized as follows. Section 2 introduces the models of computation. Section 3 investigates error-complexity tradeoffs. Section 4 presents some case studies related to the use of adaptivity. Section 5 is on a language for computing with approximations.

2 Models of Computation

In the following, we describe models of computation in which all the evaluations of arithmetic expressions are performed by arithmetic circuits. We assume for now that the computations have a fixed input-size and no recursions are involved.[1]

2.1 Circuit-oriented Model

The simplest model consists of an arithmetic circuit fed by a routing network (see Figure 3a). The input data are fed to the routing network which can compute any permutation of these data, and then provide the arithmetic circuit with the chosen permutation. The routing network is a network of comparators (see e.g. Figure 3b).

This model allows us to implement simple adaptive algorithms, as shown by the following example. This type of adaptivity will be called *off-line adaptivity*.

Example 1 *Addition of n numbers.* Consider, for example, the following adaptive algorithm for addition, which tries, by sorting the inputs, to avoid to add numbers of very different size.

[1] These aspects will be addressed in Section 2.4.

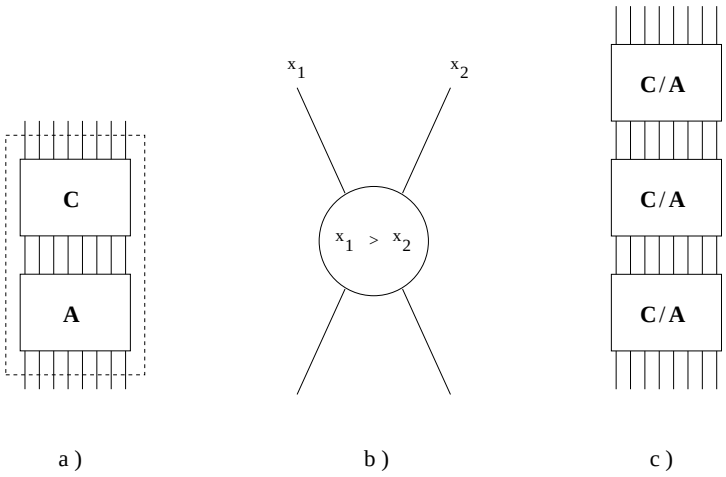

Figure 3: a) A combinatory (C) and arithmetic (A) stage.
b) The combinatory stage consists of comparators.
c) A cascade of modules. C/A is a module of the type introduced in Figure a).

Input: $a_i,\ i = 1, \ldots, n$.
1. Sort a_i in ascending order and produce $b_i,\ i = 1, \ldots, n,\ b_i = a_{\sigma(i)},\ b_i \leq b_{i+1}$, and σ is a permutation.
2. Add b_i by using the fan-in algorithm.
The resulting network is illustrated in Figure 4. ∎

A slight modification of this model consists of a cascade of "modules" of the previous kind, as shown in Figure 3c. In this case adaptivity can also be used to decide upon partial results. We will call this type of adaptivity *on-line adaptivity.*

This model allows us to implement more complicated algorithms, where adaptivity comes into play during the computation, as shown in the following example.

Example 2 *Gaussian elimination for solving linear systems.*

1. The off-line adaptive algorithm finds at the first stage, by knowing the ordering information of the numerical inputs, the optimal permutation of the input data, and then performs the elimination.

2. The on-line adaptive algorithm finds, at the i-th stage, by knowing the ordering information of intermediate numerical data, the optimal pivot for eliminating the i-th column. This approach is known as Gaussian elimination with pivoting [Vavasis, 1989].

3. The purely numerical algorithm simply performs Gaussian elimination on the original input matrix.

∎

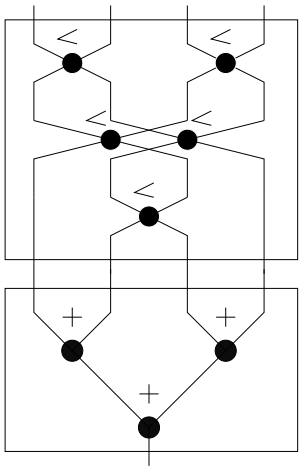

Figure 4: Adaptive addition algorithm for $n = 4$.

To describe the cost of a computation we adopt the natural measures associated to the above described models. Namely, we take, for the combinatorial and the arithmetic part the size, width and depth of the circuit, where the size is a measure of the sequential complexity (i.e. number of operations), and depth and width correspond to parallel time and number of processors respectively. Additionally we may consider the cost of one logical or arithmetic operation. In the arithmetic part we also have to analyze the error that is introduced.[2] This analysis can be performed by a "layered" error analysis, in which, for any level i in the circuit, we evaluate ϵ_i, i.e., the maximum attainable error. This technique is sketched in Section 3.

2.2 Functional Model

It is easy to see, that any arithmetic circuit corresponds in a straightforward way to an arithmetic expression. On the other hand, by also taking into account parentheses, an arithmetic expression uniquely defines a computation, which can be easily interpreted as the evaluation of an arithmetic circuit. To describe in a similar way a routing network as an expression we have to introduce *conditional functions* of the form

$$\mathrm{cond}(x_1 > x_2, x_1, x_2) = \begin{cases} x_1, \text{ if } x_1 > x_2 \text{ true}, \\ x_2, \text{ otherwise.} \end{cases} \tag{1}$$

Therefore the model described in Section 2.1 corresponds to a functional model, where the computation is described as an arithmetic expression where the variables are replaced by conditional expressions of the form (1). This is illustrated in Figure 5.

[2] In the combinatorial part no error is introduced.

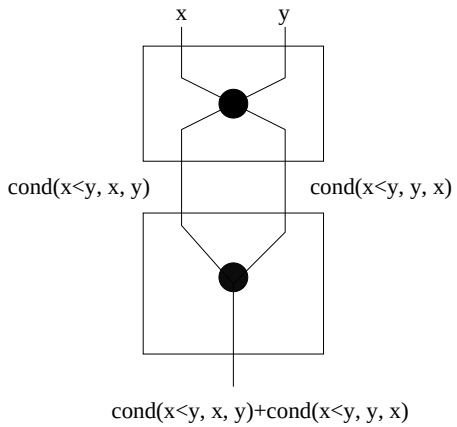

Figure 5: Translating a circuit into an expression.

So far both, the circuit and functional model, allow to use boolean information in order to compute a permutation of the inputs. In other words, in the model described here we have the fixed structure of the arithmetic circuit and the possibility of permuting the inputs. On the other hand, it could be useful to be allowed to choose more freely how to solve the problem at hand. The need for such a generalization becomes clear according to the following example.

Example 3 Assume we want to compute the function $f(x) = \sqrt{x+1} - \sqrt{x-1}$ using the arithmetic expressions $a_1 = \sqrt{x+1} - \sqrt{x-1}$, if x is "small", and $a_2 = \frac{2}{\sqrt{x+1}+\sqrt{x-1}}$ if x is "large". ∎

The model we describe in the following, based on the simple functional model introduced before, allows us to describe a numerical computation as a composition of arithmetic and particular conditional functions, whose value depends only on input data, but whose choices are among different arithmetic expressions.

Let P be a primitive predicate depending on one or more input data, e.g., $P(x) \equiv (x > 0)$ or $P(x, y) \equiv (x > y)$. Then the conditional functions are of the form

$$\text{cond}(P, e_1, e_2) = \begin{cases} e_1, \text{ if } P \text{ true}, \\ e_2, \text{ otherwise}, \end{cases} \tag{2}$$

where e_1 and e_2 are expressions built up from arithmetic and conditional functions. A numerical computation (consisting of one combinatorial and one arithmetic stage) taking k input values and computing m output values, is described by the expressions

$$e_1(x_1, \ldots, x_k), \ldots, e_m(x_1, \ldots, x_k).$$

The combinatorial stage consists of evaluating all conditional functions according to the

inputs and thus producing m arithmetic expressions $a_1, \ldots, a_m$. We denote this stage by

$$a_i = C_k^i(e_1, \ldots, e_m; x_1, \ldots, x_k),\ i = 1, \ldots, m.$$

The arithmetic stage consists of evaluating all arithmetic functions according to the input values, and producing m numerical output values. We denote this stage by

$$f_i = N_k^i(a_1, \ldots, a_m; x_1, \ldots, x_k),\ i = 1, \ldots, m.$$

Several combinatorial/arithmetic stages can be combined by composing the corresponding expressions.

Example 4 Assume we want to compute the function $f(x)$ given in Example 3 by using expression a_1, if $x \leq 1$, and a_2, when $x > 1$. The expression describing this computation is $\mathrm{cond}(x > 1, \frac{2}{\sqrt{x+1}+\sqrt{x-1}}, \sqrt{x+1} - \sqrt{x-1})$. Assume the input satisfies $x > 1$, e.g., $x = 2$. In this case, after the combinatorial stage, the arithmetic expression $\frac{2}{\sqrt{x+1}+\sqrt{x-1}}$ is returned. This expression is then numerically evaluated in the arithmetic stage. ∎

The cost of the combinatorial stage is evaluated by considering only the conditional function symbols. The cost of the numerical stage is evaluated by taking the maximum cost that can result after the execution of the combinatorial stage. In this model, the cost of performing one stage can be evaluated according to the following measures:

- *The maximum depth of all the expressions in the stage*, which is a measure related to parallel time and sequential space.
- *The maximum number of different subexpressions at one level*, which is a measure related to the number of processors and sequential time.
- *The overall size of the expressions*, which is a measure of the amount of data that has to be provided throughout the computation.

Remark. One can interpret the circuit-oriented model as an algorithm-oriented approach, where optimal (or, at least, "good") use is made of a fixed algorithm, while the functional model can be viewed as a problem-oriented approach, where the best (or, at least, a "good") algorithm is chosen to solve a problem. ∎

2.3 Data-oriented Model

The models described so far are "naive" with respect to the data-encoding issue. In this section we describe a model which is tailored to handle specific data structures. The need for such a model comes out from several arguments. As an example, consider the typical case in which adaptivity leads to sorting. All the $O(\log n)$ time parallel algorithms for sorting using only comparators are not work-optimal [Borodin & Hopcroft, 1985]. On the other hand, the two work-optimal known algorithms need to encode input data into specific data-structures [Ajtai *et al.*, 1983, Cole, 1986].

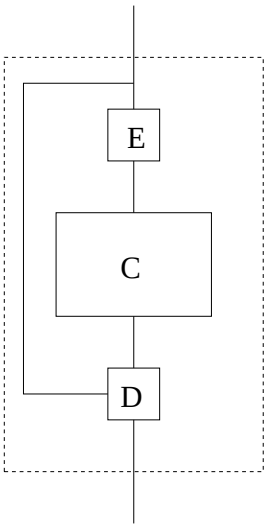

Figure 6: The data-oriented model

The model we describe here (see Figure 6), consists of building blocks corresponding to the following three phases:

1. *Encoding:* Some discrete information is extracted from the input and transferred to a discrete computation which represents the combinatorial stage.

2. *Discrete computation:* The computation produces the description of an arithmetic circuit. It may use any data structures.

3. *Decoding:* Given the description of an arithmetic circuit and the numerical input values, the computation produces the corresponding numerical result. The numerical stage corresponds to the evaluation of the arithmetic circuit.

In this model, for analyzing the cost of computations, one has to consider the additional cost occuring in the decoding and encoding phases.

2.4 Computing with Approximations

In the models we have discussed, no assumptions were made upon the data used for computations. A common assumption is that arithmetic circuits work on the real numbers model of arithmetic, i.e. one primitive arithmetic operation[3] on real numbers can be performed at unit cost. As we are mainly interested in actual numerical computations, this approach

[3] Assume that a certain (finite) set of primitive operations has been chosen, e.g., $+, -, *, \div$.

is rather unrealistic. Hence we allow the possibility to process data which are of a certain subset of the real numbers, e.g., rational numbers, arbitrary precision, fixed-point or floating-point numbers (compare Section 4). In these cases, the execution of the operations can introduce round-off errors, so that domain of the computation, and its mathematical interpretation have not to be the same.[4] To capture the real nature of numerical computations one has rather to consider the numerical values as representatives for "mathematical" data, e.g., of the form $X = (r, \epsilon)$, where r is a numerical value and ϵ is a measure of accuracy.[5] Let us consider this data as a "piece of information". Then it is natural to measure the quality of an approximation by introducing the binary approximation relation "$\sqsubseteq$"

$$X_1 = (r_1, \epsilon_1) \sqsubseteq X_2 = (r_2, \epsilon_2),$$

which has to be read as "X_2 is a better approximation (in the sense that it contains more information) than X_1". Consequently, the arithmetic operations have also to be extended to this more general domain. In the model of Section 2.1, this leads to the notion of an approximating circuit, taking approximations as inputs and returning approximations as outputs. (For a definition of approximating circuits see [Codenotti *et al.*, 1991].) In the functional model we have to consider expressions which take such approximations as inputs and return approximations as output, e.g., these expressions are of the form

$$\boldsymbol{T}(X_1, \ldots, X_k).$$

In Section 5 we describe an algebraic model, which allows us to give a meaningful interpretation for such expressions containing approximations.

2.5 Adaptivity, Recursive Algorithms, and Uniformity

In this section, we show how the structure of different types of computations reflects in our model. For simplicity of notation, we use the terminology associated to the model of Section 2.1.

1. *Purely numerical computation:* The routing network computes the trivial permutation $\sigma(i) = i$. This type of computation is oblivious.

2. *Adaptive numerical computation:* We distinguish between the two following types of adaptivity.

   (a) *Off-line adaptivity:* The whole computation consists of one combinatorial stage followed by one arithmetic evaluation.

   (b) *On-line adaptivity:* The computation consists of several combinatorial/arithmetic stages. Hence it adapts itself to intermediate numerical data.

[4] Although this is obvious, a few analyses take it seriously into account.

[5] Of course, one does not in general compute the error, which is only evaluated by an error analysis. Exceptions are, e.g., interval arithmetic computations.

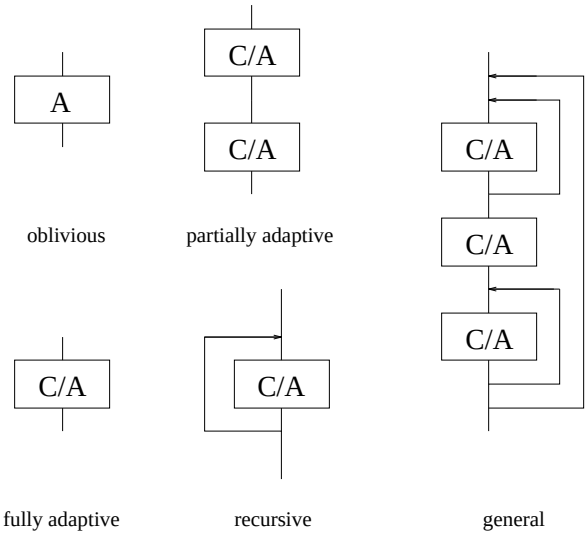

Figure 7: Different types of computation.

3. *Recursive numerical computation:* A recursive numerical computation is obtained by feeding the outputs of a combinatorial/arithmetic stage into the inputs. Also in this case, one can distinguish between adaptive and oblivious computations, depending on which stage is used.

4. *General numerical computation:* Any directed graph properly connecting combinatorial/arithmetic stages can be allowed.

Now we want to take a closer look to recursive numerical computations. If, after a finite number of steps, the numerical values reach a fixpoint, this defines the output of the recursion. This situation is not always given, e.g., the values can approach a limit or can become periodic. In this case the concept of approximation introduced in Section 2.4 turns out to be helpful, even if infinite precision is used. For the following discussion it is helpful to refer to the functional model. First we have to introduce a functional equivalent for the feed-back process, as introduced for circuits. To this extent we introduce in the simplest case (when the combinatorial/arithmetic stage is represented by an expression $\boldsymbol{T}(X)$), a new function symbol

$$\boldsymbol{M}(\boldsymbol{T}; X_0),$$

where X_0 is an input (approximation). Then the ouput of the recursion is defined as

$$\bigsqcup_{n \in \mathbb{N}} X_n,$$

where $X_{n+1} = \boldsymbol{T}(X_n) \sqcup X_n$. This approach expresses the idea that the recursion computes a (possibly) infinite sequence of better and better approximations X_n, and the result, representing the "limit", is the union of all the information contained in all components of the

sequence of approximations. For a more detailed presentation of these concepts refer to Section 5, where also general recursively defined functions are considered and examples are given.

Up to now, we have only considered the case of problems of fixed input size $(x_1, \ldots, x_{f(k)}) \in I_k$. To be able to solve classes of problems of the form $I = \bigcup_{k \in \mathbb{N}} I_k$, we have to deal with the notion of *uniformity* [Codenotti *et al.*, 1991]. For this purpose we consider the specific programs for size k as the output of a universal program for the problem which takes as input only the problem size k. Then the computation proceeds as shown above for fixed input size. We have to mention the possibility, that, in the functional model, uniformity can be achieved by using functional expressions where one of the inputs is the problem size. The functional expressions corresponding to a fixed input size are obtained by partially evaluating the expression. This approach makes it necessary to introduce tools for dealing with arbitrary long sequences of real numbers.

3 Complexity and Approximation

In numerical analysis, a central role is played by the need of evaluating how close the outcome of a sequence of rounding and approximation steps is to the exact result. Therefore a complexity theory for numerical analysis has to deal with numerical stability.

The starting point is the "conditioning" of the problem at hand. Assume we want to compute a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, $f(x_1, \ldots, x_n)$, and that the available input data are $\tilde{x_1}, \ldots, \tilde{x_n}$. Then it is worth to study the difference $f(\tilde{x_1}, \ldots, \tilde{x_n}) - f(x_1, \ldots, x_n)$, with the goal of analyzing how the differences $\tilde{x}_i - x_i,\ i = 1 \ldots, n$ carry over the function f.[6]

By using Taylor's expansion up to a first order analysis[7] we readily get

$$f(\tilde{x_1}, \ldots, \tilde{x_n}) - f(x_1, \ldots, x_n) = \sum_{i=1}^{n} (\tilde{x}_i - x_i) \frac{\partial f(x_1, \ldots, x_n)}{\partial x_i}. \tag{3}$$

The factor $\frac{\partial f(x_1, \ldots, x_n)}{\partial x_i}$ is the amplification of the error $\tilde{x}_i - x_i$, and the overall expression is called the conditioning of the computation of f.

The analysis of the conditioning of a problem, or, equivalently of a function computation, provides a lower bound on the total error in a numerical computation. In other words, it is not possible "to beat" the conditioning. Also, conditioning gives a measure of the amount of information to be carried on, in order to maintain some knowledge on the data. This is shown by the following simple example.

[6] In this section, we will carry out our analyses, using the absolute error measure. All the results can be translated onto similar results for the relative error.

[7] This is reasonable if $|\tilde{x}_i - x_i| \ll 1, i = 1 \ldots, n$

Example 5 Computation of $f(x),\ f : \mathbb{R} \to \mathbb{R}$. Let $e_x = \tilde{x} - x$.

$$|e| = |f(\tilde{x}) - f(x)| = |(\tilde{x} - x)f'(x)| = |e_x||f'(x)|.$$

It we take $|e| = 2^{-d}$, then we have

$$d \approx \log \frac{1}{|e_x|} + \log |f'(x)|.$$

∎

Note that the results related to the problem conditioning are by no means dependent on computational issues. After carrying out a perturbation analysis, the question naturally arises whether algorithms can compute a function within its perturbation error or they necessarily introduce an extra error.

This question is connected to the amount of resources the algorithm can use. Therefore the degree of adaptivity will play a central role.

A numerical process gives rise to roundoff errors. While a portion of this error could be avoided, by using "clever" algorithms, there exists another unavoidable one, which we call *representation error.*

Remark. To get an insight into the interplay between representation and perturbation error, consider the following example. Let $A(n)$ denote the problem of adding n exactly represented numbers by using binary addition as a primitive operation. It is easy to see that by rounding once, one can reduce (and this is the best one can do) $A(n)$ to $A(\frac{n}{2})$. Thus

$$\epsilon_R(n) \geq h + \epsilon_R(\frac{n}{2}) + Cond(\frac{n}{2}),$$

where $Cond(\frac{n}{2})$ denotes the conditioning of addition of $\frac{n}{2}$ numbers and $\epsilon_R(n)$ is the representation error of the addition of n numbers. ∎

Lemma 1 *Let f be a composition of primitive operations. Given a computation graph of depth t associated to the function f, the "roundoff error" ϵ_t, i.e. the difference between the actual and computed function value, can be evaluated according to the formula:*

$$\begin{aligned} \epsilon_i &= \alpha_i \epsilon_{i-1} + \epsilon_{R_i},\ i = 1, \ldots, t, \\ \epsilon_0 &= \begin{cases} \epsilon_{R_0}, & \textit{if we include the perturbation error,} \\ 0, & \textit{otherwise,} \end{cases} \end{aligned}$$

where $|\epsilon_{R_i}| \leq h$, and h is a of the system used to represent data, and the $\alpha_i,\ i = 1, \ldots, t$, are quantities depending on a primitive operation and partial results.

Proof. (Hint). In the analysis, one has to follow a path in the computation graph from the input to the output nodes. ∎

Corollary 1 *Under the assumptions of Lemma 1, if* $\epsilon_0 = 0$, *then*

$$\epsilon_t = \sum_{i=1}^{t} \epsilon_{R_i} \prod_{j=1}^{i} \alpha_j,$$

where ϵ_{R_i} *and* α_j *are as in Lemma 1.* ∎

Remark. Note that t expresses the length of a chain of subsequent roundings. ∎

Corollary 2 *Under the assumptions of Lemma 1, if* $\epsilon_0 = 0$, *then*

$$\epsilon_t = \tilde{h}t + \tilde{h} \sum_{i=1}^{t} \beta_i - \sum_{i=1}^{t} \gamma_i \beta_i - \sum_{i=1}^{t} \gamma_i,$$

where

$$\epsilon_{R_i} = \tilde{h} - \gamma_i, \text{ and } \prod_{j=1}^{i} \alpha_j = 1 + \beta_i.$$

∎

It is very difficult to express lower bounds on the error an algorithm produces, because the notion of error is of course strongly instant-dependent. Therefore we adopt the following strategy, which is based on the adversary argument (a similar approach can be found in [Codenotti *et al.*, 1991]). We assume an adversary can decide about the input distribution, so that the representation error is the roundoff error, associated to the given instance, by using the most accurate algorithm. More formally, the representation error can be expressed as

$$\epsilon_R = \min_A (\max_i \epsilon_{A(i)}),$$

where A is an algorithm, i is a problem instance, and $\epsilon_{A(i)}$ is the error occuring by executing the algorithm A with input i. Then the adversary chooses i and we choose A.

Lemma 2 *We assume that an adversary can always provide input data, for which there exists an* $\tilde{h}$, $|\tilde{h}| < h$ *such that for any path in any computation graph we have,* $\epsilon_{R_i} > \tilde{h}$, *for* $\tilde{h} > 0$, *or* $\epsilon_{R_i} < \tilde{h}$, *for* $\tilde{h} < 0$.[8] *Then, in the worst case, we have*

$$\epsilon_t = \tilde{h} \sum_{i=1}^{t} \prod_{j=1}^{i} \alpha_j = \tilde{h}t + \tilde{h} \sum_{i=1}^{t} \beta_i.$$

∎

[8] This assumption depends only on the representation and primitive functions chosen. A sufficient condition for the assumption to hold is the following. There exists a subset R of all representable numbers such that for any $\tilde{r} \in R$, we can find for any n-ary primitive function f and any $i \in \{1, \ldots, n\}$ arguments $\tilde{x_1}, \ldots, \tilde{x_n}$, such that $x_i \in R$, $\tilde{f}(\tilde{x_1}, \ldots, \tilde{x_n}) = \tilde{r}$, $f(\tilde{x_1}, \ldots, \tilde{x_n}) = r$ where $|r - \tilde{r}| \geq \tilde{h} > 0$ and always $r < \tilde{r}$ or always $r > \tilde{r}$.

We now show by an example that the above assumption on the adversary capability is reasonable.

Example 6 Consider a set of floating point numbers of the form $s * m * 2^e$, where $s \in \{-1, 0, 1\}$, $m \in \{1, \frac{5}{4}, \frac{3}{2}, \frac{7}{4}\}$ and $e \in \{-2, -1, 0, 1, 2\}$,[9] and assume that addition and multiplication are the primitive operations. In this arithmetic system the rounding is always to the closest floating point number with smaller absolute value. Then $R = \{\frac{5}{8}, \frac{7}{8}\}$ is a set of floating point numbers for which the assumption for Lemma 2 holds, with $\tilde{h} = 1/32$. This fact is shown by the following sequence of computations:

$$\frac{7}{8} * \frac{3}{4} = \frac{21}{32} = \frac{5}{8} + \frac{1}{32} \simeq \frac{5}{8} \qquad \frac{7}{8} - \frac{3}{16} = \frac{11}{16} = \frac{5}{8} + \frac{1}{16} \simeq \frac{5}{8}$$

$$\frac{5}{8} * \frac{3}{2} = \frac{15}{16} = \frac{7}{8} + \frac{1}{16} \simeq \frac{7}{8} \qquad \frac{5}{8} + \frac{5}{16} = \frac{15}{16} = \frac{7}{8} + \frac{1}{16} \simeq \frac{7}{8}$$

A similar example can also be given for fixed point arithmetic. Consider all fixed point numbers between 0 and 2 equally spaced with distance $\frac{1}{16}$.[10] Consider only multiplication as primitive operation. (Addition introduces no roundoff error in fixed point arithmetic.) Then the same choice of R provides the adversary with the power of forcing the worst case behavior. ∎

Corollary 3 *Under the hypothesis of Lemma 2, assume that the primitive operations and the intermediate data always lead to $\alpha_i \geq 1$, $i = 1, \ldots, t$. Then the representation error is in the worst case lower bounded by $|\tilde{h}|C$, where C is the longest path in the computation graph of minimum depth.* ∎

It is easy to see that, using primitive operations taking a constant number of arguments, then $C = \Omega(\log n)$, for the computation of nontrivial functions taking n arguments.

Corollary 4 *Under the hypothesis of Lemma 2, assume that the primitive operations and the intermediate data always lead to $\alpha_i \geq 1 + \epsilon$, $\epsilon > 0$. Then the representation error is in the worst case lower bounded by $|\tilde{h}|W$, where W is the width of the computation graph of minimum width.* ∎

Corollary 5 *Assume that only one primitive operation is used and $\alpha_i = \alpha > 0$, $i = 1, \ldots, t$ is independent of the intermediate data. Then we have*

$$|\epsilon_C| \geq |\tilde{h}| \; |\sum_{i=0}^{C} \alpha^i| = \begin{cases} |\frac{(1-\alpha^{C+1})}{(1-\alpha)}|, & \alpha \neq 1, \\ C\alpha, & \alpha = 1. \end{cases}$$

[9] In binary notation these are all normalized floating point numbers with mantissa of length 3 and exponent of length 2.

[10] These are all positive fixed point numbers that have a binary representation of the form $d_1.d_2d_3d_4d_5$, $d_i \in \{0, 1\}$.

Example 7 In the case of the fixed point computation of the arithmetic mean of n numbers by using the arithmetic mean of 2 numbers as a primitive operation, we have $\alpha = 1$, which gives $C|\tilde{h}|$. Note that in this case the assumptions made about the adversary hold. ∎

Theorem 1 *Under the assumption of Lemma 2, let f be a differentiable function finitely composed of the primitive operations. Let D be a lower bound on the minimal depth of a computation graph for the computation of f. Then the error in computing f by an oblivious algorithm is in the worst case at least*

$$\tilde{h}(|f'| + D),$$

provided that $\alpha_i \geq 1$. ∎

Corollary 6 *Under the hypothesis of Theorem 1, if T_P is a lower bound on the parallel time-complexity of the computation of f, then the error in computing f by an oblivious algorithm is in the worst case at least*

$$\tilde{h}(|f'| + T_P).$$

∎

Theorem 2 *Under the assumption of Lemma 2, let f be a differentiable function finitely composed of the primitive operations. Let W be a lower bound on the minimal width of a computation graph for the computation of f. Then the error in computing f by an oblivious algorithm is in the worst case at least*

$$\tilde{h}(|f'| + W),$$

provided that $\alpha_i > 1$. ∎

Corollary 7 *Under the hypothesis of Theorem 2 if T_S is a lower bound on the sequential time-complexity of the comptutation of f, then the error in computing f by an oblivious algorithm is in the worst case at least*

$$\tilde{h}(|f'| + T_S).$$

∎

Corollary 8 *Let A be an $n \times n$ nonsingular matrix and consider the matrix inversion problem. Then we have:*

$$|\epsilon| \geq \tilde{h}\|A^{-1}\| + \log n,$$

where ϵ is the absolute error and $\|\cdot\|$ stands for any matrix norm. ∎

Remark. From Corollary 8 it follows that even if A is very well conditioned, e.g. $\|A\|\,\|A^{-1}\| = O(1)$, nevertheless the error grows at least as $\log n$. It follows that an oblivious algorithm together with fixed (e.g. $O(1)$) precision can not guarantee a bounded error solution even for a very well conditioned matrix. ∎

Remark. The above lower bounds, can be translated into lower bounds on the length of the representation of the arithmetic data necessary to obtain a given error bound on the result. In fact, if we want the error ϵ to be $\epsilon \leq 2^{-d}$, then we have lower bounds on d, which depend on the problem conditioning and the complexity. ∎

The interplay between error and complexity can be further explored especially in the light of the computational cost of adaptivity. Some case studies are reported in the next section.

4 Oblivious vs. Adaptive Algorithms

In this section we first point out some basic facts about approximate computations, then we describe some case studies showing the importance of adaptivity.

4.1 Preliminaries

This section has the goal of showing the basic properties of the most popular finite representation systems.

Consider the following set of points $S^d = \{nh, n \in \mathbb{Z}\}$ of the real line, where h is a positive real number. This set can be considered as an infinite set of *fixed-point* numbers. Note that the above set is closed under addition, but it is not closed under multiplication (provided that h is not an integer). This corresponds to the well-known fact that fixed-point addition does not introduce roundoff errors, while fixed-point multiplication does (see Figure 10 on page 21).

We restrict the set S^d to a finite segment $[-M, M]$ of the real line $S^{fix} = \{nh,\ n \in \mathbb{Z} \text{ and } |nh| \leq M\}$. We now want to show some features of the usual arithmetic operations performed on elements of S^{fix} (see Figure 8 on page 19). It is easy to see that it may happen that the result of the addition of two elements of S^{fix} does not belong to S^{fix} (an overflow occurs).

It should be clear that addition and multiplication performed on a finite set of fixed point numbers have the following properties.

1. If $a + b \in [-M, M]$, then $a + b$ can be computed exactly. On the other hand, if $a * b \in [-M, M]$, then rounding errors can occur.

2. Overflow can occur both for addition and multiplication.

Analogously, by first discretization and then finitization of the real line, we can get a set S^{float}, which correspond to a generic set of floating point numbers,[11]

$$S^{float} = \{n_i\}, \text{ where}$$
$$n_0 = 0,\ n_1 = M,\ n_2 = -M,$$
$$n_{i+1} = hn_{i-1},\ i = 2, \ldots, t,\ h < 1.$$

Note that rounding errors, overflow, together with the additional phenomenon of underflow can occur, both for addition and multiplication. As it can be derived from Figure 9 on page 20 underflow is a consequence of finitization. In this case, different methods to proceed in a computation were discussed in [Demmel, 1984].

The properties of floating-point addition and multiplication can be summarized as follows.

1. If $a + b, a * b \in [-M, \frac{-M}{2^t}], [\frac{M}{2^t}, M]$ or $a + b, a * b$ are equal to 0, then the result can be represented in S^{float}, eventually by rounding.
2. Overflow and underflow can occur.

4.2 Case Studies

We would like now to show some features of numerical computations, which are usually hidden by the perturbation error associated to the most used primitive operations, i.e. addition and multiplication.[12]

Case Study 1. Consider as primitive operation the arithmetic mean. This operation is very well conditioned, i.e. the error associated to its arguments is not amplified, and the operation does not produce overflow.

Nevertheless, in Figure 11 on page 22, a situation is depicted where, besides the expected perturbation error, an additional unavoidable error occurs. This is exactly the error we have introduced in Section 3 as *representation error*. ∎

Case Study 2. Note that in [Brent, 1976] the phenomenon of representation error was incidentally discovered. In fact, it was observed that an algorithm for the computation of n digits of π in $O(t)$ steps needed to work with extra $O(\log t)$ digits. It is easy to see that this requirement was due to the unavoidable presence of what we have called here representation error. ∎

We now investigate how the representation error affects the cost of computing.

[11] The floating point numbers used in practice are indeed a combination of the set S^{fix} (mantissa) and S^{float} (exponent), as introduced here.

[12] Addition and multiplication are the most popular "basic" operations. In this paper, we use the term "primitive operations" to denote the functions which correspond to a single computation step.

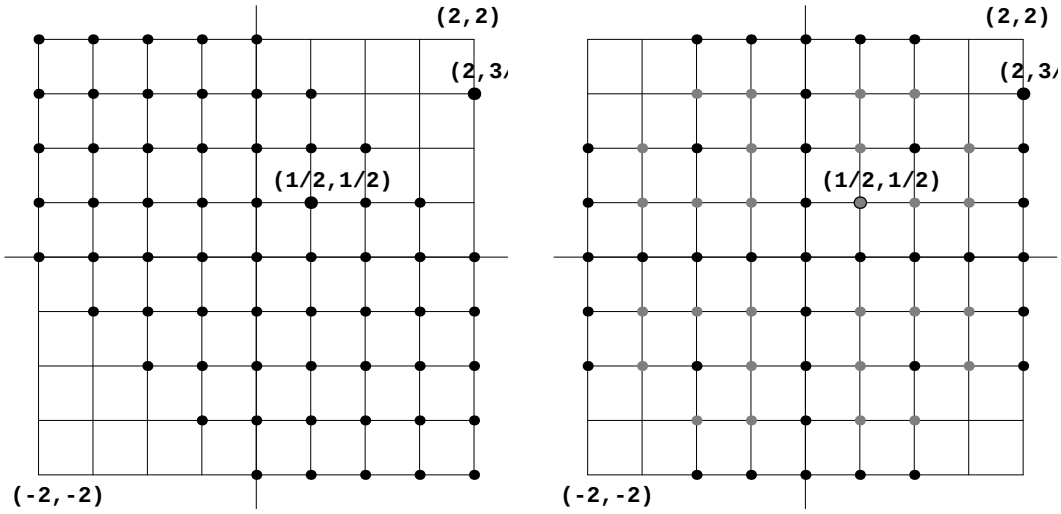

Figure 8: In this figure the pairs of fixed point numbers defined by $h = \frac{1}{2}$ and $M = 2$ are shown. They are divided, on the left side for addition and on the right sight for multiplication, into different subsets according to where the result belongs. Black points indicate, that the result belongs to S^{fix}, grey points indicate, that the result is in $[-M, M]$, but has to be rounded and, finally, no points indicate that the result does not belong to $[-M, M]$. Two typical examples are the pairs $(\frac{1}{2}, \frac{1}{2})$ and $(2, \frac{3}{2})$, which are evidenced by fatter points. For the first pair, addition gives rise to an element of S^{fix}, namely $\frac{1}{2}$, while multiplication gives $\frac{1}{4}$ and makes rounding necessary. For the pair $(2, \frac{3}{2})$ addition as well as multiplication lead to overflow.

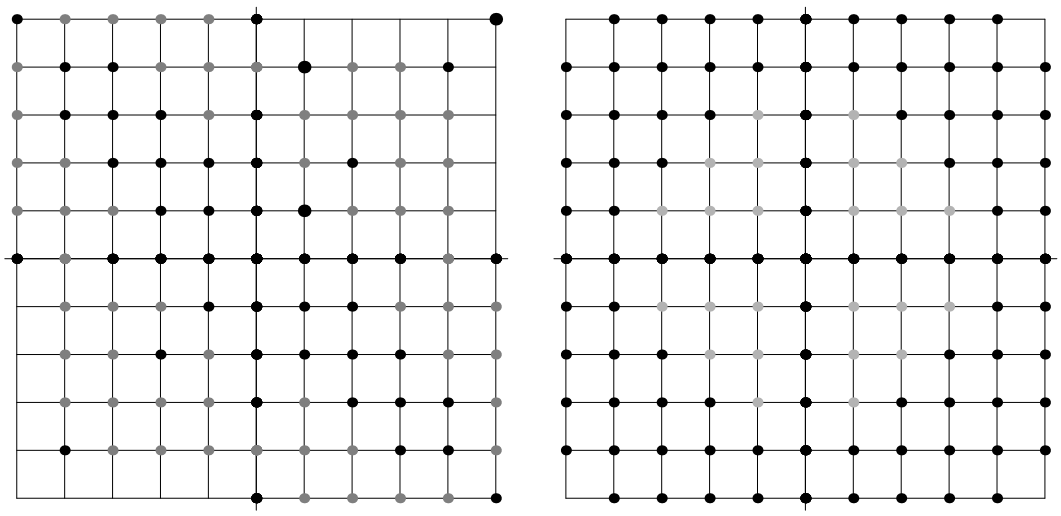

Figure 9: In this figure the pairs of floating point numbers defined by $\frac{h=1}{2}$, $M = 2$ and $t = 4$ are shown. They are divided, on the left side for addition and on the right sight for multiplication, into different subsets according to where the result belongs. Black points indicate, that the result belongs to S^{float}, dark grey points indicate, that the result is in $[-M, M]$, but has to be rounded (this occurs only for addition), light grey points indicate underflow (this occurs only for multiplication) and, finally, no points indicate that the result does not belong to $[-M, M]$. Three typical examples are the pairs $(\frac{1}{8}, \frac{1}{8})$, $(2, 2)$ and $(1, \frac{1}{8})$, which are evidenced by fatter points. For the first pair, addition gives rise to an element of S^{fix}, namely $\frac{1}{4}$, while multiplication gives $\frac{1}{64}$, which means underflow . For the pair $(2, 2)$ addition as well as multiplication lead to overflow. For the pair $(1, \frac{1}{8})$ addition makes rounding necessary, while multiplication gives an element of S^{float}, namely $\frac{1}{8}$.

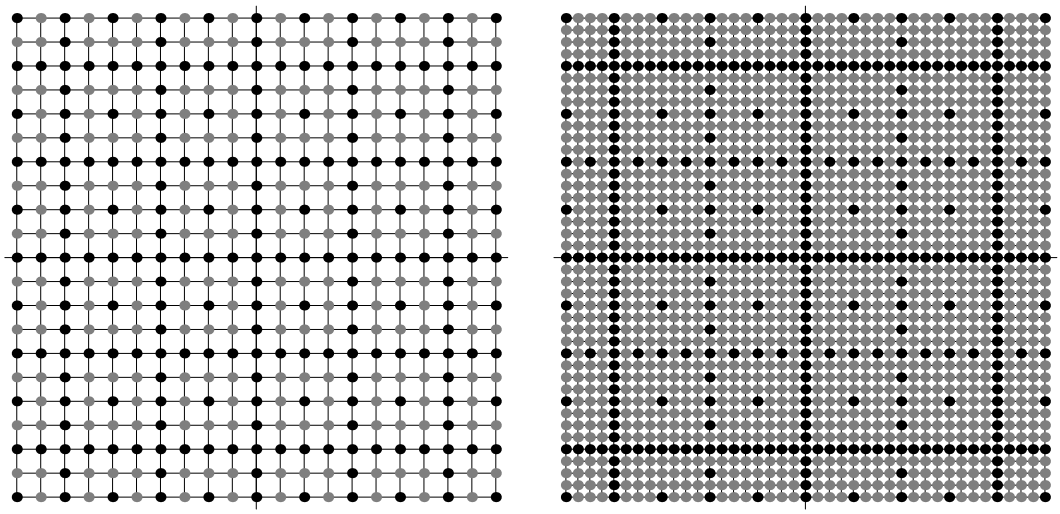

Figure 10: These pictures show pairs of numbers (a,b), $a,b \in S^d$ with $|a|,|b| \leq 5$ and $h = \frac{1}{2}$ on the left, and $h = \frac{1}{4}$ on the right. Black points mark pairs for which the result of multiplication $a * b$ is exactly another fixed point number, while grey points mark pairs for which the result has to be rounded

Case Study 3. We choose fixed-point arithmetic and consider a primitive binary operation $f(x,y)$. We assume that this operation does not amplify the initial perturbation error, i.e.,

$$|e_a \frac{\partial f(a,b)}{\partial a} + e_b \frac{\partial f(a,b)}{\partial b}| \leq \max(|e_a|,|e_b|) = e,$$

Examples of such primitive operations are the arithmetic and geometric mean. We compute a function $F(x_1,\ldots,x_k)$ as a composition of primitive operations. For example, the arithmetic mean of $n = 2^k$ numbers, namely

$$\frac{1}{n}(a_1 + \ldots + a_n),$$

could be computed from the arithmetic expression built up by the following recursion using as primitive operation the arithmetic mean of two numbers $f(x,y) = \frac{1}{2}(x+y)$:

$$f^m_{i*m} := f(f^{m-1}_{i*m}, f^{m-1}_{i*m+1}),\ f^1_i := x_i,\ i = 1,\ldots,2^{(k-m)},\ m = 2,\ldots,k.$$

The global error e_F occurring in the evaluation of this arithmetic expression is (in the worst case) proportional to the local error e_f of the evaluation of f, which is simply the rounding error. More exactly, we have

$$|e_F| = depth(F) * e,$$

where $depth(F)$ denotes the length of the longest path in the tree defined by the expression used to compute F. For the computation of the arithmetic mean of n numbers, the depth

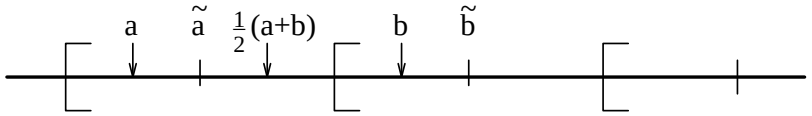

Figure 11: Let two numbers a, b be given, that are rounded to the consecutive fixed-point numbers $\tilde{a}$ and $\tilde{b}$. Thus the rounding error of the input data is at most $\frac{h}{2}$. Then the mean of $\tilde{a}$ and $\tilde{b}$ is rounded to $\tilde{b}$, while the closest fixed-point number to the actual mean of a and b is rounded to $\tilde{a}$. Hence, although the analysis of the perturbation error leads to an error of at most $h/2$, the overall error is h, and this is the best one can expect.

is $k = \log_2 n$. The global cost c_F is expressed in terms of the local cost c_f, according to the formula

$$c_F = size(F) * c_f,$$

where $size(F)$ is the number of primitive operations used in the expression computing F, and c_f is the cost to execute one primitive operation. Assume that c_f is proportional to the number b of bits used for the internal representation of numbers:

$$c_f = c_f(b).$$

In absence of perturbation error, the error of one primitive operation is

$$e_f = e_f(b) = 2^{-b+d},$$

where the constant d is the number of digits before the decimal point. Then, up to a first order analysis, we have

$$\frac{c_f(b) - c_f(|\log e_f(b)|)}{c'_f(b)} = const,$$

i.e., there is an invariant (for the variable b) in the evaluation of F. The proof of this fact is very simple, i.e.

$$c_f(|\log e_f(b)|) = c_f(b-d) \approx c_f(b) - dc'_f(b).$$

∎

Now we analyze the role of adaptivity, w.r.t the issue of numerical accuracy. We analyze some numerical and computational properties of adaptive algorithms, which behave in a very different way compared to oblivious algorithms. More precisely, we show how adaptivity can reduce the roundoff error, and we quantify the correspondent increase in computational cost. Recall that we distinguish between *off-line adaptive* algorithms, which first perform combinatorial computations on the data using any information that can be possibly derived from the inputs,[13] and then decide for a numerical algorithm and execute it, and *on-line adaptive* algorithms, which interchange combinatorial and numerical stages, and may use in the combinatorial part only information about already computed numerical data.

[13] For example compute numerically partial results.

Case Study 4. For the problem of adding n numbers we consider the following two difficulties:

- Reduce the roundoff error.
- Avoid overflow.

First we show how minimizing the roundoff error using adaptive algorithms affects the parallel time. Let us consider a floating point representation, as introduced in Section 4, and addition as primitive operation. Then assume that the sum of the following numbers

$$\frac{M}{2^t}, \frac{M}{2^t}, \frac{M}{2^{t-1}}, \frac{M}{2^{t-2}}, \ldots, \frac{M}{2^2}, \frac{M}{2},$$

has to be computed. The exact sum is M, but the only way to achieve this result is to sequentially add all the numbers from left to right. Therefore a fully adaptive algorithm which computes the optimal result, using the above optimal "selection" of the operands, would take parallel time n, whereas the optimal parallel time for this problem is $\log n$.

For avoiding overflow, we have the following.

- Oblivious algorithms: They take parallel time $\log n$ and sequential time n, but may lead to overflow, although the result lies in the segment of representable numbers.
- Partially adaptive algorithms: By sorting the arguments at each step they can always avoid overflow, if this is possible, but at the price of increasing the sequential time to $n \log n$. Parallel time is not affected.
- Fully adaptive algorithms: They can perform at least as partially adaptive algorithms. They may need more that only information about the ordering of the inputs, i.e. they need partial results. There exists a weak fully adaptive algorithm using only the ordering information about the arguments, which can always avoid overflow (if possible), but the price is (in the worst case) an $O(n)$ parallel time of size.

∎

Case Study 5. The most popular way to solve linear systems is Gaussian elimination, or, equivalently, LU-decomposition. When Gaussian elimination is performed without any re-arrangement of the rows and/or columns of the matrix, then it may give rise to a zero (very small) "pivot", so that the process cannot continue (is numerically unstable).

Thus Gaussian elimination is usually accomplished in combination with a "searching strategy", called pivoting[14] which consists of finding a permutation matrix which allows to continue the elimination or, more in general, to perform a numerically stable elimination.

[14] We should distinguish between partial and total pivoting (see, e.g. [Vavasis, 1989]).

Gaussian elimination with pivoting is a classical example of how adaptivity can be used to avoid large errors. A quantitative analysis of the advantages of such an approach can be found in [Wilkinson, 1963]. On the other hand, the "combinatorial" work needed to seek the pivot leads to a computational overhead. This extra work is asymptotically negligible in sequential models of computation, but increases the parallel time by an $O(\log n)$ factor. More fundamentally, it has been proved by Vavasis [Vavasis, 1989] that Gaussian elimination with pivoting is P-complete. This result shows that it is unlikely (i.e., unless $P = NC$) that Gaussian elimination with pivoting has an NC implementation. Since the basic Gaussian elimination process can be parallelized (see [Codenotti & Leoncini, 1991]), it follows that adaptivity, in this case, translates into a difficulty to parallelize.

From the above discussion, it follows that three basic strategies with different features can be adopted to perform Gaussian elimination (see also Example 2).

1. Oblivious algorithm: perform the elimination. This method it can be parallelized, but it can introduce huge errors. Also a division by zero can occur.
2. Fully adaptive algorithm: find a suitable permutation matrix (combinatorial stage), apply the elimination process to a permutation of the original matrix (numerical stage). The combinatorial stage can be computationally very expensive (at least P-complete), but the numerical stage can be parallelized. The process is numerically stable.
3. Partially adaptive algorithm: perform Gaussian elimination with partial or total pivoting. The process is P-complete and numerically stable.

5 An Algebraic Framework for Analytic Computations

In the previous sections several constructions turned out to be useful for describing numerical computations. Among these where conditional functions, approximations and recursion. In this section we show how these concepts fit naturally in the framework of combinatory differential fields. [Aberer, 1991]

5.1 Representation

Let us assume that a *structure*, e.g., $\mathbb{R}$, is given. To be more precise, we have to distinguish between the *domain* (e.g., $\mathbb{N}$) and the structure (e.g., $< \mathbb{N}, +, 0 >$ or $< \mathbb{N}, +, *, 0, 1 >$). In the following, S will denote the domain and $\underline{S}$ the structure. We want to compute not only with elements of this structure, but also with sets. The reason behind this choice is that we want to compute with approximations. So, when it is not possible to compute (or represent) an object exactly, we can give a set describing this object e.g., this set can be an interval to which this object belongs.

Summarizing, we are given a structure, elements of this structure, and sets.

Set representation It is too general to represent explicitly arbitrary sets and to compute with them. For example, if the underlying structure is $\mathbb{R}$, then it is impossible to represent its subsets and it is useless to deal with all the subsets. We choose to describe sets in the usual way, as done in classical analysis, namely by giving properties that specify them.

Example 8 The set Z of all the zeros of a polynomial p

$$Z = \{x : p(x) = 0\}$$

or the set of the functions F, defined by

$$F = \{f : f'(0) = 1\},$$

would be sets, that are specified by their properties. ∎

We allow only descriptions with an easy verification mechanism, in order to exclude those descriptions which use very implicit properties.

Example 9 An example of a description which is difficult to verify would be

$$\{y : \exists x \ f(x, y) = 0\}.$$

On the other hand the following description is easy to verify

$$\{y : f(y) = 0\}.$$

∎

We use a concise notation for dealing with sets. For example the set $\{y : f(y) = 0\}$ will be denoted by $\{f(@) = 0\}$, where the meaning of the symbol @ should be intuitively clear.

The descriptions introduced above must be given using a certain language. This language has to represent all the relevant operations and relations among elements of the structure. The language should also allow to express logical connectives, e.g. *and*, *or*.

Structure of the Logical Language A *logical language* consists of terms and formulas.

Terms can be atomic or composed. *Atomic terms* are the *constant* and *variable symbols*. *Composed terms* are built up by means of *operation symbols* applied to other terms.

Formulas can also be atomic or composed. *Atomic formulas* are equations of terms or predicate symbols applied to terms. *Composed formulas* are built up from other formulas using the propositional connectives *and*, *or*, *not* and the quantifiers $\exists$ and $\forall$. A formula that contains no quantifiers is called *quantifier-free*. It is easy to verify in a structure atomic formulas and also boolean compositions of them. Hence quantifier-free formulas are considered as the formulas that are easy to verify.

Example 10

Terms:	Constant symbols:	$0, 1, \iota$ (identity function).
	Variables:	$x_1, \ldots, x_k, \ldots$.
	Composed Terms:	$x + 1$.
Formulas:	Atomic formulas:	$1 = 3$, $x + 7 > 0$.
	Composed formulas:	$(x > a) \wedge (x < b)$.

∎

We now want to introduce the notion of *formula-set* in some language. Given any language, we always extend it by adding the constant symbol @. A formula-set is a set of quantifier-free formulas of this extended language.

5.2 Operations on Formula-sets

We want first to show how to compute new formula-sets starting from existing ones. An operation is defined by a term of the logic language containing free variables.

Example 11 The binary addition operation is defined by the term $x_1 + x_2$. ∎

It is clear what it means to execute this operation on elements of the underlying structure. It is also clear how to extend the notion of operation to arbitrary sets.

Example 12 Given two sets X and Y, then

$$X + Y = \{z : x \in X, y \in Y, z = x + y\}.$$

∎

But we are given sets as formula-sets and this makes it impossible to use directly the notation of Example 12. It is now convenient to introduce some notations.

Given a structure $\underline{S}$ (e.g. $\underline{S} = \mathbb{R}$) and a formula ϕ, then $\underline{S} \models_v \phi$ means that the formula ϕ holds in $\underline{S}$, under the assignment of variables given by v. In other words, the formula ϕ is satisfied in $\underline{S}$ when replacing the free variables by elements of $\underline{S}$ according to the mapping v, which assigns to every variable x_i and element $v(x_i) = \bar{x}_i \in S$. For a formula-set X we write $\underline{S} \models_v X$ if for each formula $\phi \in X$ it holds that $\underline{S} \models_v \phi$.

The notation $\phi \models^{\underline{S}} \psi$ means that, whenever the formula ϕ is satisfied in the structure $\underline{S}$ under a variable-assignment v, then the formula ψ is also satisfied in $\underline{S}$ under this variable-assignment. Or more formally, for all variable-assignments v the following holds: if $\underline{S} \models_v \phi$ then $\underline{S} \models_v \psi$. This relation can again be extended to a relation between formula-sets in the obvious way.

Let now X be a formula-set. With the notation $X|^x_{@}$ we mean that every appearance of @ is substituted by x in X. The usual set notation $\bar{x} \in \bar{X}$ can be translated to the notation

$\underline{S} \models X|^x_{@}$, where $\bar{X}$ is a set of the structure and X is a formula-set describing this set. This translation can be understood as follows: we say that an element $\bar{x}$ belongs to a set $\bar{X}$ if and only if the following condition is satisfied. All formulas of a formula-set X are true for this element. Therefore the set $\bar{X}$ is completely characterized by this formula-set.

The relation between $\bar{X}$ and X should become more clearly if one analyzes the following definitions, assuming that $v(x) = \bar{x}$:

$$\bar{X} = \{\bar{x} \in S : \text{ exists } v, \underline{S} \models_v \phi(x), \text{ for all } \phi(@) \in X\} \quad (4)$$

$$X = \{\phi(@) : \underline{S} \models_v \phi(x), \text{ for all } v \text{ s.t. } \bar{x} \in \bar{X}\} \quad (5)$$

We now show how to deal with operations on formula-sets. If we want to express $X+Y$, the following question arises: Given a variable-assignment v such that $\underline{S} \models_v X|^x_{@}$ and $\underline{S} \models_v Y|^y_{@}$, what are the properties of the sum of $\bar{x}$ and $\bar{y}$?

We want to characterize all the properties of $\bar{x} + \bar{y}$. In other words, we look for the formulas ϕ such that

$$\underline{S} \models_v \phi(x + y), \quad (6)$$

or, equivalently,

$$X|^x_{@}, Y|^y_{@} \models^{\underline{S}} \phi(x + y). \quad (7)$$

We can now conclude by giving a complete definition of addition on the structure $\underline{S}$.

$$X +_{\underline{S}} Y = \{\phi(@) : X|^x_{@}, Y|^y_{@} \models^{\underline{S}} \phi(x + y)\}. \quad (8)$$

5.3 Provability

The relation $\models^{\underline{S}}$, which is a relation on formulas or formula-sets is very general and powerful. It is also non-constructive in general. Since we are interested in computing, we have to use more restricted relations. For this purpose, logicians have introduced a more mechanical way to give relations on formula-sets, namely provability. Provability is nothing else than a mechanism which allows to derive a formula by applying certain well defined proof steps.

The most popular proof-mechanism that was devised is first order predicate logic. Appendix **A** contains the description of some basic features of first order logic, which are necessary to understand the rest of this paper.

The symbol $\vdash$ is used to denote derivations in first order logic. For example, $\phi \vdash \psi$ means that ψ can be derived from ϕ in first order predicate logic. Again this can also be extended to a relation on sets of formulas.

A *theory* is a set of sentences, which are assumed to be true. Given a theory T, the notation $\vdash^T$ denotes derivation made according to the theory T. A theory allows to describe the structure we are dealing with.

If one takes as a theory T the set of all first order sentences that are true in a given structure $\underline{S}$, which means that $\underline{S} \models T$, then this theory describes the structure, and is called the theory of the structure.

We are interested in tractable (e.g. of feasible kind) theories, namely theories such that all the sentences of the theory are provable using only a very simple (e.g. finite) set of sentences, which are called the axioms. As a consequence, the notion of derivation is feasible, since one can use the axioms (instead of the whole theory) to make derivations.

We now want to describe the relation between provability and consequence, i.e. between the symbols $\vdash^T$ and $\models^{\underline{S}}$. For the nonrelativized versions of provability and consequence the relations $\vdash$ and $\models$ are equivalent in pure first order logic. The implication $\vdash \Longrightarrow \models$, called soundness, is obviously true, and the reverse is true by the completeness theorem of first order logic. On the other hand, if we consider provability with respect to a theory, there in general only $\vdash^T \Longrightarrow \models^{\underline{S}}$ is true under the assumption that $\underline{S} \models T$.

Now we are ready to replace the relation $\models^{\underline{S}}$ by $\vdash^T$. The drawback of this replacement relies in the fact that we may loose some of the consequences. On the other hand, the advantage is that we have replaced a non-constructive notion by a constructive one.

We can now introduce the notion of logical closure of formula-sets. Given a formula-set, its closure contains all the formulas which are consequences of formulas in the formula-set. Formula-sets are said to be equivalent if they give rise to the same sets under closure. Thus, it is possible to split formula-sets into equivalence classes, each with the same closure. For this reason we will only consider logically closed formula-sets. It is natural to choose the closure as the canonical representant of each class. We also can interpret any other formula-set as representant of its class.

5.4 Definitions

We are based on a logical language L. The notation $A_@$ is used to denote the set of all formula-sets built by means of quantifier-free formulas containing the constant symbol @ and no variables.

Let T be a theory in L. Then $\boldsymbol{Cn}$ denotes the *logical closure operation* on $A_@$ with respect to $\vdash^T$. The set of logically closed formula-sets is denoted by $\mathcal{E}_{A_@}$. The elements of the set $\mathcal{E}_{A_@}$ will be called *combinators*. We will refer to combinators by giving formula-sets whose closures are the combinators.

The notation $Te(x_1, \ldots, x_k)$ denotes the set of terms of the language L containing the free variables $x_1, \ldots, x_k$.

Let now $\tau(x_1, \ldots, x_k)$ be a term of $Te(x_1, \ldots, x_k)$ denoting an operation. Then the combinatory operation corresponding to this term is defined as

$$\boldsymbol{T}^{\tau(x_1,\ldots,x_k)}(X_1, \ldots, X_k) := \{\phi(@) : X_1|_@^{x_1}, \ldots, X_k|_@^{x_k} \vdash^T \phi(\tau(x_1, \ldots, x_k))\},$$

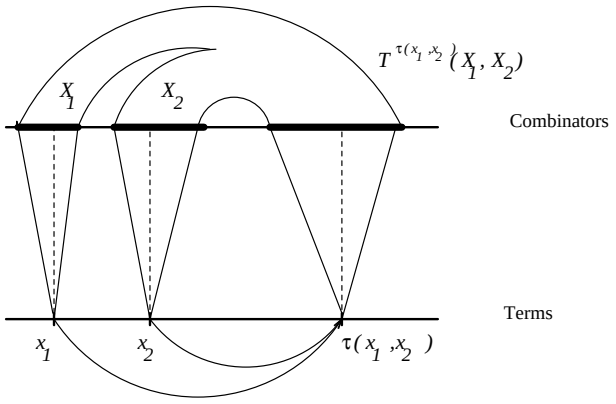

Figure 12: Combinatory operations

where $X_i, i = 1, \ldots, k$, are formula-sets in $\mathcal{E}_{A_@}$.

5.5 Lattice Structure of $\mathcal{E}_{A_@}$

An element of the underlying structure can also be interpreted as a set. An element that is described by a term can represented by the combinator $\{@ = \tau\}$. An approximation of an element can be expressed as partial knowledge on it. Partial knowledge means that we have not the whole information on the combinator at our hand. Equivalently we can say that the approximation X of the element τ is simply a subset of the formulas of the combinator describing τ.

More formally, we write

$$X \sqsubseteq \{@ = \tau\}, \tag{9}$$

to denote approximation. Definition (9) is very general; in fact it does not only apply to combinators of the type $\{@ = \tau\}$, but to all combinators. We say that a combinator X approximates a combinator Y if $X \sqsubseteq Y$.

Assume that two combinators X and Y are given. One could ask what is the knowledge they share, and if a combinator does exist which contains both the knowledge of X and Y.

The knowledge they share is simply their set-theoretic intersection, denoted as $X \sqcap Y$. The smallest formula-set which contains the knowledge of both X and Y is the set-theoretic union, but the union is not always a combinator as the next example shows.

Example 13

$$X = \{@ = \tau_1\},\ Y = \{@ = \tau_2\},\ \tau_1 \neq \tau_2.$$

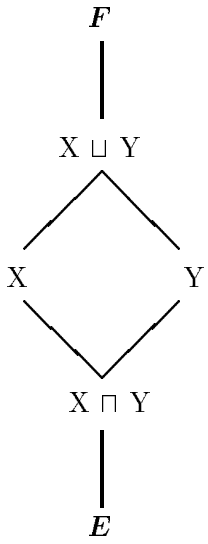

Figure 13: Lattice structure

The set-theoretic union, i.e. $\{@ = \tau_1\} \cup \{@ = \tau_2\}$ is not logically closed. In fact we have

$$@ = \tau_1, @ = \tau_2 \vdash^T \tau_1 = \tau_2 \tag{10}$$

which means that the formula $\tau_1 = \tau_2$ is a consequence of (10). It is obvious that it can not be contained in X or Y. Also observe that $X \sqcup Y$, as defined above, is simply the set $A_@$. ∎

So we have to take the logical closure of this union, and denote it by $X \sqcup Y$.

Combinators equipped with $\sqcap$ and $\sqcup$ give rise to a lattice structure, as shown in Figure 13. Appendix **B** contains the definition of lattices.

Since $\boldsymbol{F} = A_@$ is the maximum of the lattice, and $\boldsymbol{E} = \{@ = @\}$, which is contained in all combinators due to the logical closure, is the minimum, then we have a complete lattice.

5.6 Recursion

The notion of recursion naturally arises when dealing with finite or infinite sequences of combinators. In the following, we will consider monotone increasing sequences of combinators. Before introducing recursion we want to motivate the restriction to monotone increasing sequences in the computation of limits of sequences of combinators, by means of the next two examples.

Example 14 Assume we want to compute some digits of the decimal representation of a limit $a = .d_1d_2d_3 \ldots d_n \ldots$, by means of a sequence of the type $a_i = .d_1d_2d_3 \ldots d_{t_i}$, $t_i < t_{i+1}$. The sequence $\{a_i\}$ is obviously monotonically increasing as a sequence of real numbers, but

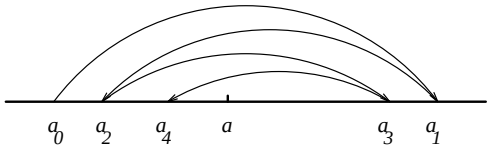

Figure 14: Nonmonotonic sequence of real numbers

there is a much more fundamental notion of monotonicity involved here. We can consider a_i not only as a representation of a real number, but also as a representation for the following information:

$$A_i = \{\text{the first } t_i \text{ digits of } @ \text{ are } d_1 d_2 d_3 \ldots d_{t_i}\}.$$

The sequence $\{A_i\}$ is monotonic of a completely different nature, namely it is a monotonic increasing sequence of information, written $A_i \sqsubseteq A_{i+1}$. ∎

Example 15 Also when the sequence is not monotonic as a sequence of real numbers it can represent a sequence of monotonically increasing knowledge. Figure 14 shows this situation, where the sequence $\{a_i\}$ is not monotonic, but the sequence $\{A_i\}$, defined as

$$A_i = \begin{cases} (a_{i-1}, a_i), \text{if } i \text{ is odd}, \\ (a_i, a_{i-1}), \text{if } i \text{ is even}, \end{cases}$$

is monotonic. ∎

One-dimensional Recursion We will provide a mechanism to produce recursive definitions of combinators. A *combinatory recursion* is given by

$$X_{n+1} = \boldsymbol{T}(X_n),$$

with starting value X_0.

In order to assure that the sequence X_n is monotonically increasing, we consider only recursions of the form

$$X_{n+1} \quad = \quad \boldsymbol{T}(X_n) \sqcup X_n. \tag{11}$$

From (11) the inclusion $X_n \sqsubseteq X_{n+1}$ easily follows. Recursion (11) defines a new combinator obtained by taking the limit

$$\bigsqcup_{n \in \mathbb{N}} X_n.$$

We view this way of producing new combinators as an operation, taking as arguments the starting value X_0 and parametrized by the operation $\boldsymbol{T}$. This operator will be denoted by

$$\boldsymbol{M}(\boldsymbol{T}; X_0).$$

Summarizing, recursion allows us to introduce the new operation $\boldsymbol{M}(\boldsymbol{T}; X_0)$, where

$$\begin{aligned} \boldsymbol{M}(\boldsymbol{T}; X_0) &= \bigsqcup_{n \in \mathbb{N}} X_n \\ X_{n+1} &= \boldsymbol{T}(X_n) \sqcup X_n, \ X_0 \text{ given.} \end{aligned} \tag{12}$$

Multi-dimensional Recursion Multi-dimensional recursion is denoted by

$$\boldsymbol{M}(\boldsymbol{T}, \boldsymbol{T}_1, \ldots, \boldsymbol{T}_k; X_0, X_0^1, \ldots, X_0^k) = \bigsqcup_{n \in \mathbb{N}} X_n.$$

As in the one-dimensional case, the sequence $\{X_n\}$ is monotone increasing and is computed using auxiliary sequences $\{X_n^i\}$, which are not necessarily monotonic, as follows.

$$\begin{aligned} X_{n+1} &= \boldsymbol{T}(X_n, X_n^1, \ldots, X_n^k) \sqcup X_n, \\ X_{n+1}^i &= \boldsymbol{T}^i(X_n^1, \ldots, X_n^k), \ i = 1, \ldots, k. \end{aligned} \tag{13}$$

5.7 Conditional Operations

These kind of operations are obviously used to make decisions in programs. The basic definition of a conditional operation is

$$\boldsymbol{C}^{\phi(x)}(X, Y, Z) = \begin{cases} Y, \text{ if } \phi(@_1) \in X, \ X \neq \boldsymbol{F}, \\ Z, \text{ if } \neg\phi(@_1) \in X, \ X \neq \boldsymbol{F}, \\ Y \sqcap Z, \text{ otherwise, if } \ X \neq \boldsymbol{F}, \\ Y \sqcup Z, \text{ otherwise.} \end{cases} \tag{14}$$

A comment on definition (14) is worthwhile. The first two cases, where X allwos to make a decision, are clear. The outcome $X \sqcap Y$ is understood by refering to Figure 13. If X does not allow a decision, there are two cases. The first is when we don't know enough, and then both alternatives are still possible. Anything between $\boldsymbol{E}$ and $Y \sqcap Z$ could be chosen as an answer in this case. According to an "optimistic" viewpoint, we choose $Y \sqcap Z$ (i.e. the maximum extractable knowledge) as an answer.

In the last case, for which $X = \boldsymbol{F}$, the condition contains so much knowledge, that one has to choose between $Y \sqcup Z$ and $\boldsymbol{F}$.

This way to introduce conditionals is useful, e.g., to handle exceptions in computations, such as for example overflow or underflow. In such a case the computations does not necessarily become trivial.

If a combinatory operation emerging from the underlying structure is approximated by the basic conditional operation given above, we will call this also a conditional operation.

5.8 Properties of Combinatory Operations

Continuity Continuity is the most basic property of operations that transform information. We first define continuity for the unary case. Let X_n be a monotone increasing chain of combinators and let $\boldsymbol{T}$ be an unary combinatory operation. Then

$$\boldsymbol{T}(\bigsqcup_{n\in\mathbb{N}} X_n) = \bigsqcup_{n\in\mathbb{N}} \boldsymbol{T}(X_n).$$

In the general case, let $X_n^1, \ldots, X_n^k$ be monotone increasing chains, and let $\boldsymbol{T}$ be a k-ary combinatory operation. Then

$$\boldsymbol{T}(\bigsqcup_{n\in\mathbb{N}} X_n^1, \ldots, \bigsqcup_{n\in\mathbb{N}} X_n^k) = \bigsqcup_{n\in\mathbb{N}} \boldsymbol{T}(X_n^1, \ldots, X_n^k).$$

A simple consequence of continuity is *monotinicity.*

$$X_1 \sqsubseteq X_2 \rightarrow \boldsymbol{T}(X_1) \sqsubseteq \boldsymbol{T}(X_2).$$

Fixed point properties of recursion The main reason for restricting the definition of recursion to the case where the main recursion sequence is monotonically increasing, is to be able to prove, using continuity, algebraic relations for recursion combinators, namely fixed point properties. In the unary case we have

$$\boldsymbol{M}(\boldsymbol{T}; X) = \boldsymbol{T}(\boldsymbol{M}(\boldsymbol{T}; X)).$$

This property can be generalized as follows. We use the shorthand notation

$$\boldsymbol{M} = \boldsymbol{M}(\boldsymbol{T}, \boldsymbol{T}^1, \ldots, \boldsymbol{T}^k; X, X^1, \ldots, X^k).$$

Let $\boldsymbol{G}$ be an m-ary combinatory operation. If $X_{n+1} = \boldsymbol{G}(X_n, \ldots, X_{n-m})$ for $n \geq m$ then

$$\boldsymbol{M} = \boldsymbol{G}(\boldsymbol{M}, \ldots, \boldsymbol{M}).$$

Embedding theorem The term structure as given by the underlying theory is isomorphically represented in the term structure of combinatory operations. This is due to the following property. Let $\tau_1, \ldots, \tau_k$ be variable-free terms. Then

$$\boldsymbol{T}^{\tau(\tau_1,\ldots,\tau_k)} = \boldsymbol{T}^{\tau(x_1,\ldots,x_k)}(\boldsymbol{T}^{\tau_1}, \ldots, \boldsymbol{T}^{\tau_k}).$$

When building up a variable-free term, one applies operations to simpler variable-free terms. Informally this theorem says is that one can interchange embedding and composition.

Soundness Assume that two operations $\tau_1, \tau_2 \in Te(x_1, \ldots, x_k)$ are given. Then

$$\tau_1(x_1, \ldots, x_k) = \tau_2(x_1, \ldots, x_k) \rightarrow \boldsymbol{T}^{\tau_1(x_1,\ldots,x_k)} = \boldsymbol{T}^{\tau_2(x_1,\ldots,x_k)}.$$

Completeness The converse of soundness is not a general property of combinatory operations but it can be proved for certain term classes. An example of such a class are terms of the theory of differential fields, which will be introduced later, containing free variables and built up by using the constants $0, 1$ and the operations $+, -, *, ^{-1}, '$.

Lifting and weakening Certain algebraic relationships involving terms with free variables can be lifted to combinatory operations. Let $\tau(x), \tau_i(x) \in Te(x), i = 1, \ldots, k$, and $\sigma(x_1, \ldots, x_k) \in Te(x_1, \ldots, x_k)$ be given and $X_1, \ldots, X_k \in \mathcal{E}_{A_{@}}$. Then

$$\begin{aligned} \boldsymbol{T}^{\tau(x)}(\boldsymbol{T}^{\sigma(x_1,\ldots,x_k)}(X_1,\ldots,X_k)) &= \boldsymbol{T}^{\tau(\sigma(x_1,\ldots,x_k))}(X_1,\ldots,X_k), \\ \boldsymbol{T}^{\sigma(x_1,\ldots,x_k)}(\boldsymbol{T}^{\tau_1(x)}(X_1),\ldots,\boldsymbol{T}^{\tau_k(x)}(X_k)) &= \boldsymbol{T}^{\sigma(\tau_1(x_1),\ldots,\tau_k(x_k))}(X_1,\ldots,X_k). \end{aligned}$$

In general such liftings lead to a loss of information, the so called *weakening*. Let

$$\tau_1(x_1, \ldots, x_k) = \tau_2(x_1, \ldots, x_1, \ldots, x_k, \ldots, x_k)$$

and $X_1, \ldots, X_k \in \mathcal{E}_{A_{@_1}}$. Then

$$\boldsymbol{T}^{\tau_2(x_1,\ldots,x_1,\ldots,x_k,\ldots,x_k)}(X_1,\ldots,X_1,\ldots,X_k,\ldots,X_k) \sqsubseteq \boldsymbol{T}^{\tau_1(x_1,\ldots,x_k)}(X_1,\ldots,X_k)$$

Diagonalization Recursions can be diagonalized, if the underlying theory allows the representation of natural numbers. For example all theories of extensions of $\mathbb{Q}$ allow such a representation.

Conditional operations Let $\boldsymbol{T}^{\tau(x)}$ be an unary combinatory operation. Then

$$\boldsymbol{T}^{\tau(x)}(\boldsymbol{C}^{\phi(x)}(X, Y, Z)) = \boldsymbol{C}^{\phi(x)}(X, \boldsymbol{T}^{\tau(x)}(Y), \boldsymbol{T}^{\tau(x)}(Z)).$$

5.9 Retractions

Combinators represent objects in a very general way. So, if one wants to consider only specific types of representations, it is convenient to introduce an operation that maps the combinator of general type to the specific type. Such a mapping has the property of a retraction.

$$\boldsymbol{R}(\boldsymbol{R}(X)) = \boldsymbol{R}(X).$$

There are two basic possibilities to introduce such retractions. The first introduces changes in representation at the level of formula-sets by restricting the formulas allowed in the representation of the object to a subset $A^r_{@}$ of $A_{@}$.

$$\boldsymbol{R}(X) = \{\phi(@) : \phi(@) \in X \wedge \phi(@) \in A^r_{@}\}.$$

Example 16 A typical example for this kind of retraction is the process of rounding real numbers to floating point or fixed point numbers with a finite number of digits. Then $A^r_{@}$ would be the finite set of formulas

$$A^r_{@} = \{@ = f_i : f_i \text{ floating (fixed) point number}, i = 1, \ldots, n\}.$$

∎

Another possibility is to introduce retractions at the level of combinatory expressions.

Example 17 The combinatory expression

$$\boldsymbol{R}(X) = \boldsymbol{C}^{x>0}(X, \{@ > 0\}, \{@ \leq 0\},$$

which has four possible results, namely $\{@ > 0\}, \{@ \leq 0\}, \boldsymbol{E}$ or $\boldsymbol{F}$, is a retraction given at the level of combinatory expressions. ∎

5.10 Combinatory Differential Fields

We are now ready to introduce the notion of *combinatory model*. The combinatory model of a theory T is the algebraic structure

$$\mathbf{E}_{A_{@}} =< \mathcal{E}_{A_{@}}; \boldsymbol{T}^{\tau(x_1,\ldots,x_k)}, \boldsymbol{C}^{\phi(x)}, \boldsymbol{M}^{\boldsymbol{T},\boldsymbol{T}^1,\ldots,\boldsymbol{T}^k} > .$$

Depending on the underlying theory we can define different combinatory models. We describe now a theory that is especially interesting for analysis.

Let F be the theory of fields of characteristic 0. The language of this theory consists of the constant symbols 0 and 1, and the operation symbols $+, *$ (binary) and $-, ^{-1}$ (unary). The field theory can be axiomatized finitely. We assume that the field contains a totally ordered constant field. Note that, if this happens, then the field is necessarily of characteristic 0. The elements of the constant field are those which satisfy the constant predicate, denoted by the constant symbol const. Since we consider function fields we introduce a constant symbol ι to denote the identity function.

A function field can be extended to a differential function field by introducing the differentiation operation (written $'$), which satisfies the usual properties with respect to the field operations. For the identity function we have

$$\iota' = 1.$$

Another operation of interest in a function field is the function composition, which we introduce as an operation on its own and denote it by $\circ$. This operation satisfies the following laws.

$$\iota \circ \tau \quad = \quad \tau \circ \iota = \tau,$$

$$
\begin{aligned}
(\tau_1 + \tau_2) \circ \sigma &= \tau_1 \circ \sigma + \tau_2 \circ \sigma, \\
((\tau_1 * \tau_2) \circ \sigma = 0 \vee \tau_1 \circ \sigma * \tau_2 \circ \sigma \neq 0) &\rightarrow (\tau_1 * \tau_2) \circ \sigma = \tau_1 \circ \sigma * \tau_2 \circ \sigma, \\
(-\tau) \circ \sigma &= -(\tau \circ \sigma), \\
\tau \circ \sigma \neq 0 \rightarrow (\tau^{-1}) \circ \sigma &= (\tau \circ \sigma)^{-1}, \\
(\tau_1 \circ \tau_2) \circ \tau_3 &= \tau_1 \circ (\tau_2 \circ \tau_3), \\
\mathrm{const}(\sigma) &\rightarrow \sigma \circ \tau = \sigma, \\
(\sigma \circ \tau)' &= \tau' * (\sigma' \circ \tau),
\end{aligned}
$$

Composition allows to define a partial order relation on functions, namely

$$
\begin{aligned}
\sigma < \tau &:\equiv \forall x \ (\mathrm{const}(x) \rightarrow \sigma \circ x < \tau \circ x), \\
\sigma \leq \tau &:\equiv \forall x \ (\mathrm{const}(x) \rightarrow \sigma \circ x \leq \tau \circ x).
\end{aligned}
$$

The combinatory model of this slightly extended theory of differential fields will be called combinatory differential field.

6 Concluding Remarks

In this paper, we have developed a model for numerical computations, which allows to evaluate error-complexity relationships and to analyze oblivious vs. adaptive numerical algorithms. We have also proved a number of results on the interplay between error and complexity. These results are intended to be a first step toward the definition of a complexity theory for numerical analysis.

The main issues are summarized in the following.

- Forcing a computation to produce a result very close to the perturbation error can be too expensive.
- Forcing a computation to compute a result with performance very close to the problem complexity can produce too many errors.
- Adaptivity allows to control the different sources of errors, but affects the cost of the algorithms.

Up to now we have been able to capture only a few aspects of the quantitative influence of adaptivity. We believe that the continuation of this research has to start from the following open problems.

- Is there a way to characterize problems for which adaptivity does not help?

- Which are the properties a problem must possess in order to be a solvable by an optimal-error adaptive algorithm, without a prohibitive combinatorial part?
- Characterize problems which are tractable using a fixed number of arithmetic digits.
- Use "easy" verification to get fast, but error-bounded, parallel algorithms.
- Use different primitive operations to have more control on the numerical error.
- Quantify the cost of adaptivity.
- Analyze the cost of avoiding overflow and underflow.

References

[Aberer, 1991] Aberer, K. (1991). Combinatory Differential Fields and Constructive Analysis. *ETH-Thesis,* **9357**.

[Aberer & Codenotti, 1991A] Aberer, K., Codenotti, B. (1991). Towards a Complexity Theory of Approximation. *manuscript submitted for publication.*

[Aberer & Codenotti, 1991B] Aberer, K., Codenotti, B. (1991). Efficiency in Numerical Analysis: Oblivious versus Adaptive Algorithms. *manuscript submitted for publication.*

[Ajtai *et al.*, 1983] Ajtai, M., Komlos, J., Szemerdedi, E., (1983). An O(n log n) Sorting Network. *Proc. of 15th Annual ACM Symp. on Theory of Computing, pp. 133-139.*

[Blum *et al.*, 1989] Blum, L., Shub, M., Smale, S. (1989). On a Theory of Computation and Complexity over the Real Numbers: NP-Completeness. Recursive Functions and Universal Machines, *Bulletin of AMS, Vol.* **21**, *No. 1, pp 1-36.*

[Borodin & Hopcroft, 1985] Borodin, A., Hopcroft, J.. (1985). Routing, Merging and Sorting on Parallel Models of Computation. *Proc. of 14th Annual ACM Symp. on Theory of Computing, pp. 338-344.*

[Brent, 1976] Brent, R.P., (1976). Fast Multiple-Precision Evaluation of Elementary Functions. Journal of ACM, Vol. 23, No. 2, pp 242-251.

[Cai, Hartmanis, 1989] Cai, J., Hartmanis, J., (1989). The Complexity of the Real Line is a Fractal. *Proceedings Fourth Annual Conference on Structure in Complexity Theory, p 138-146.*

[Chaitin, 1987] Chaitin, G. J., (1987). Information, Randomness and Incompleteness. *World Scientific.*

[Codenotti *et al.*, 1991] Codenotti, B., Leoncini, M., Resta, E. (1991). Oracle Computations in Parallel Numerical Linear Algebra. *TR ICSI 91-060.*

[Codenotti & Leoncini, 1991] Codenotti, B., Leoncini , M., (1991). Parallel Complexity of Linear System Solution. *World Scientific.*

[Cole, 1986] Cole, R., (1986). Parallel Merge Sort. *27th Annual IEEE Symposium on Foundations of Computer Science, pp. 511-516.*

[Csanky, 1976] , Csanky, L., (1976). Fast Parallel Matrix Inversion Algorithms. *SIAM J. Comput., pp 117-122.*

[Demmel, 1984] , Demmel, J., (1984). Underflow and the Reliability of Numerical Software. *SIAM J. Sci. Stat. Comput., Vol. 5, No. 4.*

[Engeler, 1981A] Engeler, E., (1981). Metamathematik der Elementarmathematik. *Springer Verlag.*

[Engeler, 1990] Engeler, E. (1990). Combinatory Differential Fields. *Theoretical Computer Science* **72**, 119-131.

[Kaplansky, 1957] Kaplansky, I. (1957). An Introduction to Differential Algebra. *Paris: Hermann.*

[Smale, 1990] Smale, S., (1990)., Some Remarks on the Foundation of Numerical Analysis. *SIAM Review, Vol. 32, No. 2, pp 211-220.*

[Traub *et al.*, 1988] Traub, J.F., Wasilkowski G.W., Wozniakowski H., (1988). Information Based Complexity. *Academic Press, New York.*

[Vavasis, 1989] Vavasis, S., (1989). Gaussian Elimination with Pivoting is P-Complete. *SIAM J. Disc. Math., Vol. 2, No.3, pp. 413-423.*

[Wilkinson, 1963] Wilkinson, J.H., (1963). Rounding Errors in Algebraic Processes. *Prentice-Hall, Englewood Cliffs, NJ.*

Appendix A: First Order Predicate Logic

A.1 Logic languages and Mathematical Structures

Logic in general, and first order predicate logic in particular, are used to describe *mathematical structures*. The mathematical structures in the scope of first order predicate logic are built up as follows.

A set S.

Functions $g_i^{k_i} : S^{k_i} \to S, i = 1, 2, \ldots$, where $k \geq 0$ denotes the arity. The nullary functions are also called constants.

Relations $R_i^{k_i} : S^{k_i} \to \{\mathbf{t}, \mathbf{f}\}, i = 1, 2, \ldots$.

So we denote the structure by $\underline{\mathcal{S}} =< S; g_1^{k_1}, g_2^{k_2}, \ldots, R_1^{l_1}, R_2^{l_2}, \ldots >$.

A *logical language* L describing such a structure is settled by giving symbols for the functions and relations of the structure. Let f_i^k be the function symbols and P_i^k the relation symbols. Then the set of *terms* is recursively defined.

Atomic terms Te_0: these are either

Variable symbols $x_0, x_1, x_2, \ldots$ or

Constant symbols $f_{i_1}^0, f_{i_2}^0, \ldots$

Composed terms Te_{n+1}: these are either

Terms in Te_0 or of the form

$f_i^{k_i}(t_1, \ldots, t_{k_i}),\ k_i > 0,\ t_j \in Te_n,\ j = 1, \ldots, k_i$

The set of all terms is then $Te = \bigcup_{n \in \mathbb{N}} Te_n$.

Similarly the recursive definition of formulas is given as follows.

Atomic formulas Fo_0: these are either

Equations $t_1 = t_2,\ t_1, t_2 \in Te$ or

Predicate symbols $P_i^{k_i}(t_1, \ldots, t_{k_i}), t_j \in Te,\ j = 1, \ldots, k_i$

Composed formulas Fo_{n+1}: these are composed by using

Propositional operations: $(\phi \wedge \psi),\ (\phi \vee \psi),\ \neg\phi,\ \phi, \psi \in Fo_n$ or

Quantifiers: $\exists x_i\ \phi, \forall x_i\ \phi,\ \phi \in Fo_n$

The set of all formulas is then $Fo = \bigcup_{n\in\mathbb{N}} Fo_n$. A variable is *bound* when it only appears in quantified form. For example x_i is bound in the formula $\exists x_i\ \phi$. Otherwise a variable is called a *free variable*. A formula containing no free variables is called a *sentence*.

Terms in the logical language correspond to functions on the structure and formulas correspond to relations. The correspondance is based on a *variable-assignment* $v : \mathbb{N} \longrightarrow S$ of the free variables. This means that the variable x_i is mapped to the element $v_i \in S$. Then we can recursively define for a term its value in the structure. This is achieved by the mapping $[\ \cdot\ ]^{\underline{S}}_v$ into the structure.

$t \in Te_0$: $[x_i]^{\underline{S}}_v = v_i$, and $[f_i^0]^{\underline{S}}_v = g_i^0$.

$t \in Te_{n+1}$: $[f_i^{k_i}(t_1, \ldots, t_{k_i})]^{\underline{S}}_v = g_i^{k_i}([t_1]^{\underline{S}}_v, \ldots, [t_{k_i}]^{\underline{S}}_v)$.

Similarly the value of a formula is given by this mapping in the following way.

$\phi \in Fo_0$:

$$[t_1 = t_2]^{\underline{S}}_v = \begin{cases} 1, \text{ if} [t_1]^{\underline{S}}_v = [t_2]^{\underline{S}}_v, \\ 0 \text{ otherwise} \end{cases}$$

$[P_i^{k_i}(t_1, \ldots, t_{k_i})]^{\underline{S}}_v = R_i^{k_i}([t_1]^{\underline{S}}_v, \ldots, [t_{k_i}]^{\underline{S}}_v)$.

$\phi \in Fo_{n+1}$:

$[(\phi \wedge \psi)]^{\underline{S}}_v = \mathbf{t}$ iff $[\phi]^{\underline{S}}_v = \mathbf{t}$ and $[\psi]^{\underline{S}}_v = \mathbf{t}$. The value is given analogous for the other propositional connectives.

$[\exists x_i\ \phi]^{\underline{S}}_v = \mathbf{t}$ iff exists $u \in S$ such that $[\phi]^{\underline{S}}_v = \mathbf{t}$ for any variable-assignment v such that $v_i = u$.

$[\forall x_i\ \phi]^{\underline{S}}_v = \mathbf{t}$ iff for all $u \in S$ $[\phi]^{\underline{S}}_v = \mathbf{t}$ for any variable-assignment v such that $v_i = u$.

For a sentence the value is obviously independent of the specific variable-assignment. So a sentence can be only true or false in a structure $\underline{S}$. If it is true we write for this fact

$$\underline{S} \models \phi.$$

A sentence is a *logical identity* if $\underline{S} \models \phi$ for all structures $\underline{S}$ that can be described in the chosen language. Then we write $\models \phi$. The notation $X \models \psi$ where X is a set of sentences and ψ is a sentence denotes the following fact: For all structures for which $\underline{S} \models \phi$ for all $\phi \in X$ we have $\underline{S} \models \psi$.

A basic theorem that illustrates the relation $\models$, is the *deduction theorem*: If ϕ und ψ are sentences then $\phi \models \psi$ iff $\models \phi \rightarrow \psi$.

A.2 Provability

We come now to introduce a syntactic equivalent to the relation $\models$. We say that a formula ψ is provable from a set of sentences X, if there exists a sequence of formulas $\psi_1, \ldots, \psi_n$, such that $\phi = \psi_n$ and each of the other formulas in the sequence is either a formula out of X, a logical axiom or a formula derived from preceding formulas in the sequence according to logical inference rules. We write for this

$$X \vdash \phi$$

The logical axioms of first order predicate logic are divided in the following groups.

Propositional tautologies.

Equality axioms

$(\forall x \phi(x)) \rightarrow \phi(t),\ (\phi(t) \rightarrow (\exists x \phi(x))$

A few remarks have to be made for each of these axiom groups.

Propositional formulas are built up from *prime formulas* which can have the value **t** or **f** and the propositional connectives $\wedge, \vee, \neg, \rightarrow$. The value of the formula, given the values of the prime formulas, is evaluated in the well known way. For example if the formula is $A \wedge B$ and the value is **t** iff the value of both of A and B is true. *Propositional tautologies* are those formulas which evaluate to **t** for all possible values of its prime formulas. The set of propositional tautologies can be axiomatized by a finite axiom set and propositional tautologies can be proven out of these axioms by using *modus ponens*, a derivation rule that will also be introduced for predicate logic below.

The equality axioms are as follows:

$$\begin{aligned} t &= t \\ (t_1 = t_2) &\rightarrow (t_2 = t_1) \\ (t_1 = t_2 \wedge t_2 = t_3) &\rightarrow (t_1 = t_3) \\ (t_1 = s_1 \wedge \ldots \wedge t_k = s_k) &\rightarrow (P_i^k(t_1, \ldots, t_k) = P_i^k(s_1, \ldots, s_k)) \\ (t_1 = s_1 \wedge \ldots \wedge t_k = s_k) &\rightarrow (f_i^k(t_1, \ldots, t_k) = f_i^k(s_1, \ldots, s_k)) \end{aligned}$$

For the last two axioms the following restriction has to be satisfied. The variables bound in ϕ do not occur in t.

The inference rules are *modus ponens*

$$\frac{(\phi \rightarrow \psi) \quad \phi}{\psi}$$

and the *generalization rules*

$$\frac{\phi \to \psi(x)}{\phi \to \forall y \psi(y)}, \quad \frac{\psi(x) \to \phi}{\exists y \psi(y) \to \phi}.$$

For the latter two rules again a restriction applies. The variable x does not occur free in ϕ.

The proof system is chosen such that it is *sound*. That means if $\phi \vdash \psi$ then for any structure $\underline{\mathcal{S}}$ and variable-assignment v if $[\phi]^{\underline{S}}_v = \mathbf{t}$ then $[\psi]^{\underline{S}}_v = \mathbf{t}$.

The fundamental theorem about provability in first order predicate logic is Gödels *completeness theorem* which tells us that for sentences the above property holds in both directions: A sentence ϕ is provable from a set of sentences X iff if it is true in all models of X. Or more formally

$$X \vdash \phi \text{ iff } X \models \phi.$$

Since a proof $X \vdash \phi$ is always finite it can also make only use of a finite number of sentences of X. So we can state that if $X \models \phi$ then there exists a finite subset $X_0 \subseteq X$ such that $X_0 \models \phi$. This has the following consequence, called the *compactness theorem*: A set of sentences X is consistent iff every finite subset of it is consistent.

Appendix B: Complete Lattices, Combinatory Algebras and Graph Models

B.1 Complete Lattices

A *complete lattice* is a partial order set D, where the partial order relation is denoted by $\sqsubseteq$, which has a *minimal element* $\bot$ and where for each subset $X \subseteq D$ exists a *supremum* $\bigsqcup X$, such that $x \sqsubseteq \bigsqcup X$ for all $x \in X$. The existence of a minimum and suprema implies also the existence of a maximum $\top$ and an infimum $\bigsqcap X$ for each subset $X \sqsubseteq D$. The supremum and infimum of subsets consisting of two elements x and y are also denoted by $x \sqcup y$ and $x \sqcap y$.

There exists an algebraic characterization of the properties of $\sqcup$ and $\sqcap$. An *algebraic lattice* is the algebraic structure $< D; \sqcup, \sqcap >$ satsifying the following axioms.

$$\begin{aligned} x \sqcup y &= y \sqcup x, & x \sqcup x &= x, \\ x \sqcup (y \sqcup z) &= (x \sqcup y) \sqcup z, & x \sqcup (x \sqcup y) &= x, \end{aligned}$$

and analoguous for $\sqcap$. The relation $\sqsubseteq$ can be defined in an algebraic lattice by $x \sqsubseteq y \equiv x \sqcap y = x$.

A mapping $f : D \to D$ is *monotone* if for $x \sqsubseteq y$ it holds that $f(x) \sqsubseteq f(y)$. A point x is a fixpoint of f if $f(x) = x$. An important property of complete lattices is the theorem of

Tarski. It says that for a monotone mapping f the set of fixpoints agains forms a complete lattice.

A subset $X \subseteq D$ is directed if for $x, y \in X$ there exists always a $z \in X$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. A mapping $d : D \to D$ is *continuous* if for all directed X

$$f(\bigsqcup X) = \bigsqcup f(X).$$

The definition of $f(X)$ is $f(X) = \{f(x) : x \in X\}$. A complete lattice that contains all continuous functions on it as elements is called a *complete continuous lattice*.

B.2 Combinatory Algebras

The main idea for the introduction of combinatory algebras is to reduce arbitrary function applications to unary function applications, a process called *Currying*. To do this a binary application operation $\cdot$ is introduced. The process of Currying is then illustrated by the following example.

Example 18 Let the ternary function $f(x, y, z)$ be given. Then we rewrite this as $((f \cdot x) \cdot y) \cdot z$. So $(f \cdot x) \cdot y$ is a unary function applied to z with x and y fixed. ∎

Note that application must not be confused with function composition. Let $f(g(x))$ the composition of functions, then written using the application operation this yields $f \cdot (g \cdot x))$. The goal of combinatory algebra is to express every "composition rule" expressed as combinatory term built up by free variables and the application operation by some basic composition rules. In the case of function composition we would look for a combinator $\boldsymbol{B}$ such that $f \cdot (g \cdot x) = \boldsymbol{B} \cdot f \cdot g \cdot x$. It turns out that two basic combinators are sufficient to express all possible composition rules.

$$\begin{aligned} \boldsymbol{K} \cdot X \cdot Y &= X \\ \boldsymbol{S} \cdot X \cdot Y \cdot Z &= X \cdot Z \cdot (Y \cdot Z) \end{aligned}$$

The completeness theorem of combinatory logic says that for every combinatory term $t(x_1, \ldots, x_k)$, built up from the application operation and free variables $x_1, \ldots, x_k$ there exists a combinatory term $\boldsymbol{T}$ built up from $\boldsymbol{K}$ and $\boldsymbol{S}$ only such that

$$t(x_1, \ldots, x_k) = \boldsymbol{T} \cdot x_1 \cdot \ldots \cdot x_k.$$

Example 19 $\boldsymbol{B} = \boldsymbol{S} \cdot (\boldsymbol{K} \cdot \boldsymbol{S}) \cdot \boldsymbol{K}$. ∎

Another important property of combinatory algebra is that fixpoint equations are solvable. Given a fixpoint equation

$$\boldsymbol{F} \cdot X = X,$$

there exists a combinatory term $\boldsymbol{Y}$, expressible in $\boldsymbol{K}$ and $\boldsymbol{S}$, such that

$$\boldsymbol{F} \cdot (\boldsymbol{Y} \cdot \boldsymbol{F}) = \boldsymbol{Y} \cdot \boldsymbol{F}.$$

Combinatory algebras are able to represent all partial recursive functions on the natural numbers. There exists an encoding of the natural numbers, which assigns to each $n \in \mathbb{N}$ a combinatory term $\underline{n}$, such that for any partial recursive function $f : \mathbb{N} \to \mathbb{N}$ there exists a combinatory term $\boldsymbol{F}$ such that

$$\boldsymbol{F} \cdot \underline{n} = \underline{f(n)}.$$

B.3 Graph Models

There exist many models of combinatory algebras of which the graph model is most important to us. A domain $\mathcal{D}_A$ of the graph model $\mathbf{D}_A$ is built up on a base set A by the following recursive definition.

$$\begin{aligned} G_0(A) &= A \\ G_{n+1}(A) &= G_n(A) \cup \{\alpha \to a : \alpha \text{ finite, , } \alpha \subseteq G_n(A),\ a \in G_n(A)\} \\ G(A) &= \bigcup_{n=0}^{\infty} G_n(A), \\ \mathcal{D}_A &= \mathcal{P}(G(A)). \end{aligned}$$

($\mathcal{P}$ denoted the powerset.) The application operation for $M, N \in \mathcal{D}_A$ is given by

$$M \cdot N = \{a : \exists\ \alpha \subseteq N,\ \alpha \to a \in M\}.$$

To show that this is a model of combinatory algebra one gives combinators for $\boldsymbol{K}$ and $\boldsymbol{S}$ and verifies their properties.

On the other hand graph models are complete lattices where the lattice structure is simply induced by the partial order relation on $\mathcal{D}_A$. The universality of graph models is illustrated by the following important theorem. A mapping $f : \mathcal{D}_A \to \mathcal{D}_A$ is continuous iff there exists an element $\boldsymbol{F} \in \mathcal{D}_A$ such that for all $x \in \mathcal{D}_A$

$$\boldsymbol{F} \cdot x = f(x).$$

This shows that all combinatory operations of graph models are monotone and continuous.

For the embedding of algebraic structures into graph models the notion of *retraction* is central. A retraction $\boldsymbol{R}$ is mapping such that

$$\boldsymbol{R} \cdot (\boldsymbol{R} \cdot x) = \boldsymbol{R} \cdot x.$$

The retract $R = \{x : x = \boldsymbol{R} \cdot x\}$ is as a set of fixpoints a complete lattice. An *inner algebra* $\mathbf{A} =< \mathcal{A}; \boldsymbol{T} >$ of a graph model has a retract $\mathcal{A} = \{X \in \mathcal{D}_A : \boldsymbol{R} \cdot X = X\}$ as *carrier set* and an operation $\boldsymbol{T} \in \mathcal{D}_A$ on this carrier set satisfies $\boldsymbol{T} \cdot (\boldsymbol{R} \cdot X_1) \cdot \ldots \cdot (\boldsymbol{R} \cdot X_n) = \boldsymbol{R} \cdot (\boldsymbol{T} \cdot (\boldsymbol{R} \cdot X_1) \cdot \ldots \cdot (\boldsymbol{R} \cdot X_n))$.

We can now formulate the main theorem for *combinatory models*.

Theorem 3 *The combinatory model*

$$\mathbf{E}_{A_{\mathbb{Q}_1,\ldots,\mathbb{Q}_n}} =< \mathcal{E}_{A_{\mathbb{Q}}}; \boldsymbol{T}^{\tau(x_1,\ldots,x_k)}, \boldsymbol{C}^{\phi(x)}, \boldsymbol{M}^{\boldsymbol{T},\boldsymbol{T}^1,\ldots,\boldsymbol{T}^k} >$$

of a theory T is an inner algebra of the graph model $\mathbf{D}_{A_{\mathbb{Q}_1,\ldots,\mathbb{Q}_n}}$.

The proof of the theorem makes use of several facts exhibited above. First compactness of first order logic allows to represent the combinatory operations of the form $\boldsymbol{T}^{\tau(x_1,\ldots,x_k)}$ and $\boldsymbol{C}^{\phi(x)}$. For the representations of recursions $\boldsymbol{M}^{\boldsymbol{T},\boldsymbol{T}^1,\ldots,\boldsymbol{T}^k}$ fixpoint combinators are used. Composition of functions can be represented by combinators generalizing $\boldsymbol{B}$.