Combinatory Models and Symbolic Computation

Karl Aberer [1]

Abstract

We introduce an algebraic model of computation which is especially useful for the description of computations in analysis. On one level the model allows the representation of algebraic computation and on an other level approximate computation is represented. Furthermore programs are themselves algebraic expressions. Therefore it is possible to algebraically manipulate programs of symbolic and numerical computation, thus providing symbolic computation with a firm semantic foundation and giving a natural model for mixed symbolic-numerical computation. We illustrate these facts with examples.

1 Introduction

A substantial part of computer algebra deals with problems arising from analysis, see for example [Buchberger *et al.*, 1983, Davenport *et al.*, 1988]. This computational approach to analysis makes use of algebraic representations of analytic structures, as for example differential fields [Kaplansky, 1957]. The objects used for these algebraic computations in analysis are represented exactly. The computational approach to analysis which exploits the set-theoretic properties of structures defined over the real numbers, leads to computational methods, which are usually related to numerical computation. In this case usually approximative representations for objects are used.

Computational models for analysis which origin from recursion theory, like, e.g., recursive analysis [Pour-El & Richards, 1989] or the theory of machines on the reals [Blum *et al.*, 1989], are, as models of computation with real numbers, naturally closer to numerical computations. However, in the theory of computation there exist models of computation which are given as algebraic structures, e.g., combinatory algebras. A popular model of combinatory algebras is λ-calculus [Barendregt, 1977]. In this work we consider other models of combinatory algebras, namely graph models [Engeler, 1981A, Engeler, 1981B]. It was shown that any algebraic structure can be embedded in a graph model [Engeler, 1988]. Hence graph models give rise to an algebraic model of computation in algebraic structures. So it appears appropriate to use them as models for symbolic computation. On the other hand, graph models have, similarly as analytic structures, a second facet. They are also endowed with the structure of a complete continuous lattice [Maeder, 1986], an important structure in denotational semantics to model approximate computations. This fact was used to model numerical computations in graph models [Fehlmann, 1981]. Similar ideas for complete partial orders can also be found in [Weihrauch, 1980].

We will introduce, based on graph models, *combinatory models* as computational models for analysis. This allows to represent uniformly exact and approximate computations in analysis in an algebraic structure. The main ideas leading to this model go back to [Engeler, 1990].

This model is relevant for symbolic computation for several reasons.

[1] International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704, USA. e-mail: aberer@icsi.berkeley.edu.

- The description of symbolic programs of analysis, built up from algebraic and computational operations, including recursion, in an algebraic computational model allows to view these programs themselves as algebraic expressions. This allows to state algebraic relationships for such programs.
- The embedding allows to introduce new types of field extensions which are defined by programs. Classically in computer algebra, e.g., differential fields are extended by new transcendental functions, such as given by elementary extensions. This leads to a rich theory culminating in the results of Risch [Risch, 1969]. When also the extension by functions, which are defined by program, is possible, new powerful solution mechanisms are introduced in algebraic computation.
- Computer algebra systems usually are equipped with a substantial set of numerical methods, *mathematica* is a good example for this [Wolfram, 1988]. Numerical mathematics tends to make more use of symbolic methods, see, e.g., [Kaucher, 1983, Stetter, 1988]. The trend is leading to mixed symbolic-numeric computations. For such computations combinatory models of analytic structures provide a clear semantics and thus a well defined interface for smooth transitions between the two worlds. This will also allow to deal with programs, which compute approximately, as algebraic objects.
- Many problems of analysis have symbolic as well as numerical solution methods. A computational model that allows to describe both, helps to compare the different methods, e.g., in order to analyze the complexity of analytic problems. Only a few approaches are known that allow such a general investigation of analytic problems [Traub *et al.*, 1988].
- Last but not least a well defined semantics for computations in analysis, taking into account the different aspects arising from using algebraic, approximative and computational constructs, can give guidelines to build safer systems, especially for computer algebra but also for mathematical computation in analytic structures in general.

In this paper we will be able to go closer into some of these aspects after introducing the notion of combinatory models. The reader interested in the relations to graph models is advised to refer to [Aberer, 1991A, Aberer, 1991B].

2 Operations of Symbolic Computation

We distinguish the following three types of operations used in computer algebra. [2]

1. Algebraic operations: `D[f,x], f/g`.
2. Computational operations: `If[c>0,f,g], While[a>0, .... ]`. Also composition of programs is considered as a computational operation.

[2]We use in this paper *Mathematica* as one representative of a typical classical computer algebra system. All notations for computer algebra programs are hence taken out of it. Other system, that would have served as well, are *Maple*, *Macsyma* or *Reduce*. *Axiom* (former *Scratchpad*) would have made some of the examples given more difficult to present due to its type-checking mechanism.

3. Representation-related operations: `Expand[f], Series[f,{x,x0,n}]`.

We consider programs of symbolic computations as algebraic expressions composed from these operations. The following examples show how this viewpoint can lead to strange answers of actual systems, if done "carelessly", i.e., if no account is taken of underlying structural assumptions or features of the implementation.

Example 1 Assume we want to define in the following examples specific real functions.

1. In this example we pick out a situation, that well can occur during a computation.

```
In[19]:= f[x_]:=x/0

In[20]:= f[1]
                                 1
Power::infy: Infinite expression - encountered.
                                 0
Out[20]= ComplexInfinity

In[21]:= 0 * ComplexInfinity

Infinity::indt: Indeterminate expression 0 ComplexInfinity encountered.

Out[21]= Indeterminate
```

2. Here we make a reasonable attempt to define the absolute value function.

```
In[21]:= f[x_]:=If[x>0,x,-x]

In[22]:= D[f[x],x]

             (0,0,1)                    (0,1,0)
Out[22]= -If        [x > 0, x, -x] + If        [x > 0, x, -x] +

             (1,0)          (1,0,0)
>    Greater     [x, 0] If        [x > 0, x, -x]
```

3. Now we make a less reasonable attempt to define a function recursively.

```
In[23]:= f[x_]:=f[x]/2

In[24]:= f[1]

General::recursion: Recursion depth of 256 exceeded.

Out[24]=
                                   1
                              Hold[-] Hold[f[1]]
                                   2
>   ---------------------------------------------------------------------------
    904625697166532776746648320380374280103671755200316906558262375061821325312
```

The following is a more reasonable, but similarly successless, approach.

```
In[9]:= g[x_,n_]:=1/2 g[x,n-1]

In[10]:= g[x,0]:=x

In[11]:= g[x]:=Limit[g[x,n],n->Infinity]

In[12]:= g[1]

Out[12]= g[1]

In[15]:= D[g[x],x]

General::recursion: Recursion depth of 256 exceeded.

General::stop: Further output of General::recursion
     will be suppressed during this calculation.
^C
```

(The calculation was interrupted.)

∎

Discussion of the examples:

1. The answer is almost satisfying. Open is the question about a mathematical interpretation of `ComplexInfinity` and about the validity of the system's assumption that the result is over a complex number field.
2. The systems response to a very natural attempt to define the absolute value function is completely inappropriate. In this case a mathematical model of how to deal with piecewise defined functions is simply missing.
3. The third example illustrates that symbolic computation systems lack an algebraic mechanism to deal with functions defined as limits of infinite recursions, despite the fact, that these functions play an important role in computations in analysis.

We have to remark that these phenomena are not particular for *mathematica* but similar behavior occurs in many other systems.

The failures presented in the above examples are mainly due to the fact, that the programming constructs cannot by safely composed to new expressions, or that basic programming constructs like recursion are not provided. This again is due to the lack of an algebraic structure which contains such constructs as elements, and provides them with a consistent semantics.

3 Representation of Approximations

Now let us turn to approximative methods. We have to consider two aspects. First the results of an approximative method are often computed as limits of recursive sequences of approximations. Typical examples are Newton's method for finding zeros of a function or power-series methods for solving differential equations, where especially the latter also play an important role in symbolic computation. Therefore recursion should be provided as an algebraic construct itself. Second approximative methods make, as the name says, use of approximations. To be able to deal with approximations in an algebraic environment we need an algebraic approach to the notion of approximation and algebraic properties of approximations.

We will use the notion of *approximating an object* in the sense of having *partial knowledge of all properties of the object.*

A typical example of such partial knowledge arising from computer algebra, is found in the representation of a real algebraic number. Two kinds of partial knowledge are combined to represent a real algebraic number exactly. Let the symbol @ denote the real algebraic number. Then first a polynomial p is given of which @ is a zero: $p(@) = 0$. Second an interval $[a, b]$ is given which isolates @: $a \leq @ \leq b$. So the set of formulas $X = \{p(@) = 0, a \leq @ \leq b\}$ describes the real algebraic number completely.

We now generalize this way of representing an object (approximately or exactly). Assume an analytic structure, e.g., a totally ordered field or differential field, is defined by means of a first order theory T using a language L of first order predicate calculus with equality, not containing the special symbol @. The language L contains all the constant, operation and predicate symbols necessary for the description of the specific analytic structure. For a totally ordered field it would at least include the constant symbols $0, 1$, the operation symbols $+, -, *, ^{-1}$ and the predicate symbol $<$. We denote the set of variable-free, quantifier-free formulas in @ by $A_@$. Then we represent (partial) knowledge of an object @ of this structure by a finite or infinite set formulas X, which are satisfied by this object.

$$X = \{\phi_1(@), \phi_2(@), \ldots\} \subseteq A_@.$$

The reason to choose quantifier-free formulas is, intuitively spoken, that they are easy to verify. In the computational model we develop we will compute with such formula-sets instead of the elements of a structure. This will allow to model in an uniform way computations with approximations, by computing with formulas-sets that do not determine an object uniquely, and objects that are known exactly, e.g., by an algebraic expression τ. In this case the object is described completely by the formulas-set $\{@ = \tau\}$.

Note that this representation of knowledge gives naturally rise to a lattice structure on partial knowledge, which is simply given by the lattice induced by subset-inclusion. We will say that a formula-set X *approximates* another formula-set Y if

$$X \subseteq Y.$$

This lattice is complete since there exists a minimum, namely the empty set, and a maximum, namely $A_@$.

We have introduced three types of operations: algebraic, computational and representation-related. For each of these we will give the corresponding operations on formula-sets.

4 Algebraic operations

One can think of an algebraic operation, like `f+g`, as an operation, that is defined by an algebraic expression containing free variables. In the example of addition the algebraic expression would be $\tau(x, y) = x + y$. Now let us have a look back on the representation of real algebraic numbers. For the addition of two real algebraic numbers we have to be able to add the corresponding intervals. Intuitively it is clear how to add two intervals.

$$\{a \leq @ \leq b\} + \{c \leq @ \leq d\} = \{a + c \leq @ \leq b + d\}.$$

This can be described more formally as an operation on formula-sets as follows. If T is the theory of totally ordered fields then we can prove

$$a \leq x \leq b, c \leq y \leq d \vdash^T a + c \leq x + y \leq b + d.$$

where $X \vdash^T \phi$ means, that the formula ϕ is in first order logic provable from the set of formulas X, using theorems from T. Hence an element, being the sum of two elements satisfying $\{a \leq @ \leq b\}$ and $\{c \leq @ \leq d\}$, has to satisfy $\{a + c \leq @ \leq b + d\}$.

This leads to a definition generalizing an operation given by a term $\tau(x_1, \ldots, x_k)$, $k \geq 0$, to an operation $\boldsymbol{T}^{\tau(x_1,\ldots,x_k)}$ on formula-sets $X_1, \ldots, X_k \in A_@$. $X_i|_@^{x_i}$ denotes the set of formulas where in X_i every appearance of @ is substituted by x_i.

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

To simplify notation we will maintain the usual mathematical notations also for combinatory operations, as far as no confusion is possible, e.g., for $\boldsymbol{T}^{\tau(x,y)}(X, Y)$ we write simply $X + Y$ or for $\boldsymbol{T}^0$ we write 0.

The logical closure $\boldsymbol{Cn}(X)$ of a formula-set X is the formula-set, that contains all formulas which are logical consequences of X under $\vdash^T$. Any object satisfying all formulas in X also satisfies all formulas in $\boldsymbol{Cn}(X)$. So it makes no sense to distinguish between formula-sets with the same logical closure. We call such logically closed formula-sets *combinators* and we assume that a formula-set from now on always denotes its logical closure. We denote the set of combinators with $\mathcal{E}_{A_@}$. Note that an operation $\boldsymbol{T}^{\tau(x_1,\ldots,x_k)}$ always maps combinators into combinators. Hence we call such an operation *combinatory operation.*

The set of logically closed formula-set again forms a complete lattice, where the maximum is, as for formula-sets, $\boldsymbol{F} = A_@$, while the minimum of this lattice is now $\boldsymbol{E} = \boldsymbol{Cn}(\emptyset)$. We denote inclusion of combinators in this lattice by

$$X \sqsubseteq Y.$$

The lattice operations of finding the infimum and supremum of two logically closed formula-sets X, Y are then given by

$$X \sqcap Y = X \cap Y, X \sqcup Y = \boldsymbol{Cn}(X \cup Y).$$

5 Computational operations

The simplest computational operation, namely *composition*, is realized by composing combinatory expressions. Let, e.g., $\boldsymbol{T}_1(X), \boldsymbol{T}_2(X)$ be combinatory operations, then their composition is given by $\boldsymbol{T}_1(\boldsymbol{T}_2(X))$.

A decision function, a so-called *conditional*, can be defined as follows:

$$\boldsymbol{C}^{\phi(x)}(X_1, X_2, X_3) = \begin{cases} X_2, \text{ if } \phi(@) \in X_1,\ X_1 \neq \boldsymbol{F} \\ X_3, \text{ if } \neg\phi(@) \in X_1,\ X_1 \neq \boldsymbol{F} \\ X_2 \sqcap X_3, \text{ otherwise, if } \ X_1 \neq \boldsymbol{F} \\ X_2 \sqcup X_3, \text{ otherwise,} \end{cases}$$

where $\phi(@)$ is a formula in $A_@$ and represents the condition on which the decision is based. The two first cases are easy to understand. In the third case there is not enough information to make the decision, so both alternatives may be returned. Hence the minimal knowledge which holds for the both alternatives, namely $X_2 \sqcap X_3$, is returned. In the last case, where a contradiction occurs, both alternatives have to be returned at the same time, so one choses the best possible answer $X_2 \sqcup X_3$. This way to define decisions allows a natural exception handling by catching up computations that have failed by providing too little or too much knowledge.

We now come to the task of representing *infinite recursion.* This first requires to introduce a limit notion for infinite sequences of combinators.

Assume that a sequence of combinators $X_n,\ n \in \mathbb{N}$, is given. We define the limit of such a sequence as the union

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

i.e., as the union of the knowledge contained in all X_n.

Example 2 Let X_n be the nth Taylor approximation of $\exp(\iota)$ which we express as combinator by

$$X_n = 1 + \iota + \ldots + \frac{\iota^n}{n!} + \boldsymbol{D}_n,$$

where ι denotes the identity function, i.e., $\iota' = 1$. The combinator $\boldsymbol{D}_n = \{@(0) = 0, @'(0) = 0, \ldots, @^{(n)}(0) = 0\}$ allows to substitute the notation $o(\iota^n)$, which is usually used to denote the higher-order terms of a truncated power-series. Now $\frac{\iota^{n+1}}{(n+1)!}$ is approximated by $\boldsymbol{D}_n = \{@(0) = 0, @'(0) = 0, \ldots, @^{(n)}(0) = 0\}$. Furthermore $\boldsymbol{D}_n \sqsubseteq \boldsymbol{D}_{n+1}$. Hence

$$1 + \iota + \ldots + \frac{\iota^n}{n!} + \boldsymbol{D}_n \sqsubseteq 1 + \iota + \ldots + \frac{\iota^n}{n!} + \frac{\iota^{n+1}}{(n+1)!} + \boldsymbol{D}_{n+1}.$$

This shows that $X_n \sqsubseteq X_{n+1}$ or in other words that X_n is a monotonically increasing sequence of combinators. ∎

We consider in the following only limits of monotonically increasing sequences of combinators, so called *chains*. This is no essential restriction even when considering nonmonotonic number or function sequences as the following example shows.

Example 3 Assume the nonmonotonic sequence $a_i, i \geq 0$, of real numbers is given and converges to a limit a such that $a_i < a$ for i odd and $a_i > a$ for i even. Then we can easily construct out of this sequence a monotonic sequence of combinators, such that the limit of combinators describes the same point a, by

$$\boldsymbol{A}_i = \{b \leq @ \leq c : b = \min_{j \leq i} a_j, c = \max_{j \leq i} a_j\}^3.$$

∎

We define a combinator, which we want to be the limit of an infinite recursion, as the union of a recursively computed, monotone increasing sequence of combinators X_n. Let combinatory operations $\boldsymbol{T}, \boldsymbol{T}^1, \ldots, \boldsymbol{T}^k$ and starting values $X_0, X_0^1, \ldots, X_0^k$ be given. Then the recursion for computing the X_n is defined as follows.

$$\begin{array}{rcl} X_{n+1} & := & \boldsymbol{T}(X_n, X_n^1, \ldots, X_n^k) \sqcup X_n \\ X_{n+1}^1 & := & \boldsymbol{T}^1(X_n^1, \ldots, X_n^k) \\ & \vdots & \\ X_{n+1}^k & := & \boldsymbol{T}^k(X_n^1, \ldots, X_n^k). \end{array}$$

The $X_n^1, \ldots, X_n^k$ can be considered as auxiliary sequences, which have not necessarily to be monotonic, while the monotonicity of X_n is ensured by the inclusion $X_n \sqsubseteq X_{n+1}$, which holds for the main recursion. We denote the combinator $\bigsqcup_{n \in \mathbb{N}}$ computed by the recursion by

$$\boldsymbol{M}^{\boldsymbol{T}, \boldsymbol{T}^1, \ldots, \boldsymbol{T}^k}(X_0, X_0^1, \ldots, X_0^k).$$

So $\boldsymbol{M}^{\boldsymbol{T}, \boldsymbol{T}^1, \ldots, \boldsymbol{T}^k}$ is a new combinatory operation in the arguments $X_0, X_0^1, \ldots, X_0^k$. [4] The $\boldsymbol{T}, \boldsymbol{T}^1, \ldots, \boldsymbol{T}^k$ can be considered as parameters and to obtain better readability we will denote this operation, using arguments $X, X^1, \ldots, X^k$, alternatively by

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

6 Combinatory Models

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

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

[3] We assume that $\min(\emptyset) = -\infty$ and $\max(\emptyset) = \infty$. An inequality of the form $@ < \infty$ is then equivalent to a trivial formula, e.g. $@ < @ + 1$.

[4] In *Axiom* a similar construct is actually used for the representation of power-series.

Depending on the underlying theory T we can define different combinatory models. If T is the field theory then we have a *combinatory field*, if T is the differential fields theory then we have a *combinatory differential field.* In most cases it is useful to include in the equational theories of fields and differential fields ordering predicates, hence considering, e.g., totally ordered fields or partial ordered function fields. This allows the use of approximations which are described, e.g., as intervals.

We consider programs in analytic strucutres as algebraic expressions of combinatory models and program execution is performed by evaluating these expressions.

The following theorem is of central importance. It relates combinatory models to graph models and justifies that combinatory models are indeed algebraic structures.

Main Theorem: [Aberer, 1991B]
The combinatory model

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

is an inner algebra of the graph model $\mathbf{D}_{A_@}$.

This theorem allows us to use methods of the theory of graph models to prove properties in combinatory models. This leads to the following basic properties of combinatory operations. Proofs for these can be found in [Aberer, 1991B].

1. 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 *monotonicity.*

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

2. Fixpoint 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 fixpoint 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 $X_{n+1} = \boldsymbol{G}(X_n, \dots, X_{n-m})$ for $n \geq m$ then

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

3. 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 $\tau_1, \dots, \tau_k$ be variable-free terms. Then

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

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.

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

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

5. 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 $+, -, *, ^{-1}, '$. Some completeness results can be found in [Aberer, 1991A].

6. Lifting and Weakening: Certain algebraic relationships involving terms with free variables can be lifted into the combinatory model.

Let $\tau(x), \tau_i(x) \in Te(x), i = 1, \dots, k$, and $\sigma(x_1, \dots, x_k) \in Te(x_1, \dots, x_k)$ be given and $X_1, \dots, X_k \in \mathcal{E}_{A_@}$. Then

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

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

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

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

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

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

8. Conditional operations: Laws for terms composed of conditional and algebraic operations can be derived. Let $\boldsymbol{T}^{\tau(x)}$ be an unary combinatory operation. Then

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

7 Symbolic programs in combinatory models

We give now interpretations of the programs discussed in the examples of Section 2. Division by 0 is represented by the combinatory term $\boldsymbol{T}^{\frac{1}{0}}$. Now, all that can be proved about $\frac{1}{0}$ in a field theory are trivial properties that are true for all elements. So the correct interpretation is

$$\boldsymbol{T}^{\frac{1}{0}} = \boldsymbol{E}.$$

This seems also to be the intended interpretation of `Indeterminate` in *mathematica*. If an object related to the concept of infinity is needed, this can be provided in a combinatory model by using recursion, e.g., by

$$\boldsymbol{M}(\boldsymbol{T}^{2*x}; \{-1 \leq @ \leq 1\}).$$

The absolute value function is given as

$$\boldsymbol{A}(X) = \boldsymbol{C}^{x>0}(X, \boldsymbol{T}^{x}(X), \boldsymbol{T}^{-x}(X)).$$

Its derivative can be computed in a combinatory model as follows

$$\boldsymbol{A}'(X) = \boldsymbol{T}^{x'}(\boldsymbol{A}(X)) = \boldsymbol{C}^{x>0}(X, \boldsymbol{T}^{x'}(X), \boldsymbol{T}^{-x'}(X)).$$

Thus, e.g., $\boldsymbol{A}(0) = 0$ and $\boldsymbol{A}'(0) = -1$. In this way we can algebraically manipulate general piecewise defined functions.

Infinite recursion can be represented now in closed form. One has only to take care of the initial value, such that the sequence of approximations is monotone increasing. [5] A recursion approximating 0 is, e.g., given by the recursion operation

$$\boldsymbol{M}(\boldsymbol{T}^{\frac{x}{2}}; \{0 \leq @ \leq 1\}).$$

8 Representation-related operations

We have omitted this class of operations so far. Operations that change the representation of an object are very useful in symbolic computation. There exist operations that preserve the complete knowledge about an object, like `Expand[x]`, and operations that lead to a loss of information, like `Series[y,{x,x0,n}]`. The second type of representation-related operations are often used for transitions to numerical computation, e.g., using `N[x]` to get a floating point representation of a real number. One property all representation-related operations have in common is that they are *retractions*. A retraction $\boldsymbol{R}$ is an operation that satisfies

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

Now we show two basic possibilities to introduce such retractions in combinatory models. The first operates on the level of formula-sets by restricting the formulas allowed in the representation of the object to a subset $A^r_@$ of $A_@$.

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

[5] Note that $\boldsymbol{M}(\boldsymbol{T}^{\frac{x}{2}}; 1) = \boldsymbol{F}$.

These operations are elements of the combinatory model [Aberer, 1991A].

Example 4 A typical example for this type of retractions would be rounding of real numbers to floating point or fixed point numbers with finite precision. 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 define the retractions as combinatory expressions, which means we introduce the retraction in form of a program.

Example 5 We illustrate this by the computing the Taylor series as a monotone increasing sequence of Taylor polynomials. This can be done by the following recursive program $\boldsymbol{Ps}(X) := \bigsqcup_{n\in\mathbb{N}} \boldsymbol{P}_n$.

$$\begin{array}{lcllcl}
\boldsymbol{P}_{n+1} & := & y_n + \boldsymbol{D}_n & \boldsymbol{P}_0 & = & \boldsymbol{E} \\
y_{n+1} & := & y_n + \frac{dy_n(0)}{n!} * \iota^n & y_0 & = & X \\
dy_{n+1} & := & dy'_n & dy_0 & = & X \\
(n+1)! & := & n! * n & 0! & = & 1 \\
\boldsymbol{D}_{n+1} & := & \iota * \boldsymbol{D}_n & \boldsymbol{D}_0 & = & \{@(0) = 0\}
\end{array}$$

To show that $\boldsymbol{P}_n$ is monotone increasing we use the following three facts. [6]

1. $\boldsymbol{D}_n \sqsubseteq \boldsymbol{D}_{n+1}$: Follows by monotinicity from $\boldsymbol{D}_0 \sqsubseteq \iota * \boldsymbol{D}_0$, which is shown by $(\iota * y)'(0) = \iota'(0) * y(0) + \iota(0) * y'(0) = 1 * y(0) + 0 * y'(0)$.
2. $\boldsymbol{D}_n \sqsubseteq \iota^{n+1}$: Follows by monotinicity from $\boldsymbol{D}_0 \sqsubseteq \iota$.
3. $\boldsymbol{D}_n = \boldsymbol{D}_n + \boldsymbol{D}_n$: It is clear that $\boldsymbol{D}_0 = \boldsymbol{D}_0 + \boldsymbol{D}_0$. Using the fact that for a combinator $\boldsymbol{T}^\tau$ representing a term τ we have $\boldsymbol{T}^\tau * (X + X) = \boldsymbol{T}^\tau * X + \boldsymbol{T}^\tau * X$, we conclude by induction on n that $\boldsymbol{D}_n + \boldsymbol{D}_n = \iota * \boldsymbol{D}_{n-1} + \iota * \boldsymbol{D}_{n-1} = \iota * (\boldsymbol{D}_{n-1} + \boldsymbol{D}_{n-1}) = \iota * \boldsymbol{D}_{n-1} = \boldsymbol{D}_n$.

This allows us to conclude

$$\boldsymbol{P}_n = y_{n-1} + \boldsymbol{D}_{n-1} = y_{n-1} + \boldsymbol{D}_{n-1} + \boldsymbol{D}_{n-1} \sqsubseteq y_{n-1} + \frac{dy_n(0)}{n!} * \iota^n + \boldsymbol{D}_n = \boldsymbol{P}_{n-1}.$$

The recursive operation also defines a retraction. This is shown by using continuity:

$$\boldsymbol{Ps}(\bigsqcup_{n\in\mathbb{N}} \boldsymbol{P}_n) = \bigsqcup_{n\in\mathbb{N}} \boldsymbol{Ps}(\boldsymbol{P}_n),$$

and remarking that $\boldsymbol{Ps}(\boldsymbol{P}_n) = \boldsymbol{P}_n$. ∎

[6] Note that these are also properties of $o(*\iota^n)$.

9 Symbolic Derivation of Approximate Algorithms

We want to give an illustration how the properties of combinatory operations can be used to derive symbolically approximative algorithms. The goal is to compute approximative solutions of linear differential equations, which can be represented in closed-form by using recursion operators. Although the idea of power-series plays in the following a central role, we have to point out, that the algorithm not only gives the correct solution up to a given order, but also gives inclusions of the solution in form of function intervals. This means the algorithm computes a substantial additional amount of information, which is especially useful to obtain verified bounds on the solution. Similar techniques play a role in numerical computations, see e.g., [Weissinger, 1988] for inclusion methods for differential equations.

We consider the case of a regular, linear, homogenous differential equation over the real numbers with polynomial coefficients, which can be written in the form

$$L(y) = y^{(m)} + a_{m-1}(\iota) * y^{(m-1)} + \ldots + a_1(\iota) * y' + a_0(\iota) * y = 0.$$

It is a well known fact that the solution of such an equation is analytic in a neighborhood of any point. Furthermore the coefficients of the power-series expansion of the analytic function are computable by a polynomial recursion. We will use this to construct a combinatory solution of the differential equation, in the form of a monotone increasing and recursively computed chain of approximations of the solution.

First we give the recursion for computing the power-series expansion in a way, that is especially suited for our purposes. Let $L^e(y)$ be $L(y)$ written in expanded form, i.e., the derivatives $y^{(i)}$ are distributed over the polynomials $a_i(\iota)$. [7] Then substitute every monomial of the form

$$c * \iota^k * y^{(i)} \quad \text{by} \quad c * \iota^k * y^{(i)}_{n+(i-m)-k},$$

where c is a constant coefficient and $y_n, \ldots$ are new variables. The y_n will turn out to be the power-series of the solution truncated at the n-th power. This substitution gives an operator

$$\widetilde{L}(y_n, \ldots, y_{n-t}),$$

where

$$t = \max_{i=1,\ldots,m-1} deg(a_i) - (i - m).$$

We make the ansatz

$$y_{n+1} = y_n + u_n,\ u_n = b_n * \iota^n,\ b_n \text{ constant}, \tag{1}$$

for the recursion and assume that

$$\widetilde{L}(y_n, \ldots, y_{n-t}) = 0. \tag{2}$$

Note that $\widetilde{L}(y, \ldots, y) = L(y) = L^e(y)$ and

$$\widetilde{L}(y, \ldots, y) \equiv L^e(y).$$

[7]This means that $L(y) = L^e(y)$ but $L(y) \not\equiv L^e(y)$, where $\equiv$ denotes syntactic equivalence. This will be of importance in the combinatory embedding later.

If we substitute in $\widetilde{L}$ according to (1) and use the fact that $\widetilde{L}$ is linear y_i, the equation (2) is satisfied iff

$$\widetilde{L}(u_n, \ldots, u_{n-t}) = 0.$$

This implies that the u_n satisfy a condition of the form

$$u_n = \sum_{i=0}^{t-1} p_i(n) * \iota^i * u_{n-i-1},$$

where $p_i(n)$ are polynomials in n. This gives the desired recursive computation of y_n. The starting values $u_0, \ldots, u_{t-1}$ have to be determined according to the initial values of the differential equation.

Up to now we have only reformulated well known facts. The crucial step now is to make a combinatory ansatz. We want to recursively compute an increasing chain of approximations X_n of the form

$$X_n := y_{n-t} + V_1(n) * u_{n-1} + \ldots + V_t(n) * u_{n-t} = y_{n-t} + \sum_{i=1}^{t} V_i(n) * u_{n-i}, \tag{3}$$

where $V_i(n)$ are combinators with the property $\text{const}(@) \in V_i(n)$. We substitute this ansatz into $\widetilde{L}$ and then use the following combinatory laws which follow from the properties given for internal combinators (C is a constant combinator, i.e., $\text{const}(@) \in C$):

$$(X+Y)' = X' + Y',\ (C * X)' = C * X',\ C * X + C * Y \sqsubseteq C * (X+Y).$$

Using linearity in $\widetilde{L}$ we get

$$\begin{aligned}\widetilde{L}(X_n, \ldots, X_{n-t}) &= \widetilde{L}(y_{n-t}, \ldots, y_{n-2t}) + \widetilde{L}(V_1(n) * u_{n-1}, \ldots, V_1(n) * u_{n-t-1}) + \ldots \\ &\qquad + \widetilde{L}(V_t(n) * u_{n-t}, \ldots, V_t(n) * u_{n-2t}) \\ &\sqsubseteq V_1(n) * \widetilde{L}(u_{n-1}, \ldots, u_{n-t}) + \ldots V_t(n) * \widetilde{L}(u_{n-t-1}, \ldots, u_{n-2t}) = 0.\end{aligned}$$

If we assume that the X_n are a monotone increasing chain then we may conclude the following by using continuity.

$$\bigsqcup_{n \in \mathbb{N}} \widetilde{L}(X_n, \ldots, X_{n-t}) = \widetilde{L}(\bigsqcup_{n \in \mathbb{N}} X_n, \ldots, \bigsqcup_{n \in \mathbb{N}} X_n) = L^e(\bigsqcup_{n \in \mathbb{N}} X_n) \sqsubseteq 0. \tag{4}$$

Observe how we made use of the syntactical equivalence of $\widetilde{L}(y, \ldots, y)$ and $L^e(y)$, such that no weakening occurred in this step. Hence (4) implies that, if $L^e(\bigsqcup_{n \in \mathbb{N}} X_n)$ is convergent in the sense that two elements satisfying all formulas contained in this combinator are arbitrarily close, then $\bigsqcup_{n \in \mathbb{N}} X_n$ is a unique solution.

Now we want to investigate under which circumstances the sequence X_n is a chain. This will also make clear why the ansatz for X_n was chosen exactly as in (3). To do this we make the assumption that we are only interested in the solution on a certain interval $\boldsymbol{C} = \{a \leq @ \leq b\}$, where $-1 \leq a \leq b \leq 1$. This will allow us to use the inclusion $\boldsymbol{C} \sqsubseteq \iota^k$. Then we get

$$
\begin{aligned}
X_{n+1} &= y_{n-t+1} + \sum_{i=0}^{t-1} V_{i+1}(n) * u_{n-i} \\
&= y_{n-t} + u_{n-t} + \sum_{i=0}^{t-1} V_{i+1}(n) * u_{n-i} \\
&= y_{n-t} + u_{n-t} + V_1(n) * \sum_{i=1}^{t} p_i(n) * u_{n-i} * \iota^i + \sum_{i=1}^{t-1} V_{i+1}(n) * u_{n-i} \\
&= y_{n-t} + u_{n-t} * (1 + V_1(n) * p_t(n) * \iota^t) + \sum_{i=1}^{t-1} u_{n-i} * (V_{i+1}(n) + V_1(n) * p_i(n) * \iota^i) \\
&\sqsupseteq y_{n-t} + u_{n-t} * (1 + V_1(n) * p_t(n) * \boldsymbol{C}) + \sum_{i=1}^{t-1} u_{n-i} * (V_{i+1}(n) + V_1(n) * p_i(n) * \boldsymbol{C})
\end{aligned}
$$

The X_n form a chain if the last line is an approximation of

$$X_n = y_{n-t} + V_1(n) * u_{n-1} + \ldots + V_t(n) * u_{n-t} = y_{n-t} + \sum_{i=1}^{t} V_i(n) * u_{n-i}.$$

This is the case if the following system of inclusions is satisfied.

$$
\begin{aligned}
V_t(n-1) &\sqsubseteq 1 + V_1(n) * p_t(n) * \boldsymbol{C} \\
V_i(n-1) &\sqsubseteq V_{i+1}(n) + V_1(n) * p_i(n) * \boldsymbol{C}, \; i = 1, \ldots, t-1.
\end{aligned}
$$

These inclusions are satisfied if the corresponding equations are satisfied. The corresponding system of equations looks like a linear system of equations. In fact if we assume that $\boldsymbol{C}$ is a real number and the V_i are independent of n we have a linear system of t equations in t unknowns, which (in general) [8] is nondegenerate. We leave it as an open question whether the combinatory system is solvable in general for any linear differential equation, although this seems to be very likely following the above arguments.

We illustrate the algorithm for a concrete example which was computed by using *mathematica.* Take the differential equation

$$y''' + y' + x * y = 0, \; y(0) = 1, \; y'(0) = 1, \; y''(0) = 2.$$

The recursion is computed as

```
             4                     2                2
            j  u[-4 + n]       2 j  u[-2 + n]    j  n u[-2 + n]
u[n] -> -(---------------) + --------------- - ---------------
                      2    3               2    3                2    3
            2 n - 3 n  + n     2 n - 3 n  + n    2 n - 3 n  + n
```

The equation to be satisfied are

[8]It is easy to compute in this case the determinant and to see that only for exceptional values of $\boldsymbol{C}$ and n the system degenerates.

```
{V[2, n] == V[1, -1 + n],

     Int[-1, 0] V[1, n]
>    ------------------ + V[3, n] == V[2, -1 + n], V[4, n] == V[3, -1 + n],
         (-1 + n) n

         Int[-1, 0] V[1, n]
>    1 + ------------------- == V[4, -1 + n]}
         (-2 + n) (-1 + n) n
```

One solution of this system is

```
V[1,n_]:=Int[0,1];
V[2,n_]:=Int[0,1];
V[3,n_]:=Int[1-1/(n+2)/(n+1)/n,1];
V[4,n_]:=Int[1-1/(n+1)/(n-1)/n,1];
```

The first few iterates are then as follows.

```
          3       1         2                  5             23
X[3]    = J  Int[-(-), 0] + J  Int[0, 1] + Int[-, 1] + J Int[--, 1]
                  6                            6             24



              3       1         4       1              23        2     59
X[4]    = 1 + J  Int[-(-), 0] + J  Int[-(-), 0] + J Int[--, 1] + J  Int[--, 1]
                      6                 8              24              60

                  3       1     119      4       1         5        1
X[5]    = 1 + J + J  Int[-(-), -(---)] + J  Int[-(-), 0] + J  Int[-(---), 0] +
                          6     720              8                 120

                  2    3       1     119      4       1     209
X[6]    = 1 + J + J  + J  Int[-(-), -(---)] + J  Int[-(-), -(----)] +
                               6     720              8     1680

      5       1          6       1
>    J  Int[-(---), 0] + J  Int[-(---), 0]
             120                240
                       3
                  2   J     4       1     209       5        1       67
X[7]    = 1 + J + J  - -- + J  Int[-(-), -(----)] + J  Int[-(---), -(----)] +
                       6              8     1680              120     8064

      6       1          7       1
>    J  Int[-(---), 0] + J  Int[0, ----]
             240                 1008
```

Note that the iterates are monotone increasing combinators and how the sizes of the intervals shrink.

10 Conclusion

We have presented an algebraic approach to deal with approximate computation. This allows us to use the *qualitative* properties of approximations in an algebraic computing environment. Additionally this approach includes other important concepts for computing in analysis, like conditional functions, infinite recursion and mechanisms for exception handling. The *quantitative* properties of approximation, which lead to questions of convergence and complexity, are part of ongoing work [Aberer & Codenotti, 1992], and are important steps toward further integration of concepts from numerical and symbolic computation.

References

[Aberer, 1990] Aberer, K. (1990). Normal Forms in Function Fields. *Proceedings ISSAC '90*, 1-7.

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

[Aberer, 1991B] Aberer, K. (1991): Combinatory Differential Fields: An Algebraic Approach to Approximate Computation and Constructive Analysis. *TR-91-061, International Computer Science Institute, Berkeley.*.

[Aberer & Codenotti, 1992] Aberer, K., Codenotti, B. (1992). Towards a Complexity Theory of Approximation. *TR-92-012, International Computer Science Institute, Berkeley.*

[Barendregt, 1977] Barendregt, H. (1977): The type free lambda calculus. *Handbook of Mathematical Logic, ed. Jon Barwise, North Holland.*

[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.*

[Buchberger *et al.*, 1983] Buchberger, B., Collins, B., Loos, R. (1983).
Computer Algebra-Symbolic and Algebraic Computation. *Springer, Wien, New York.*

[Davenport *et al.*, 1988] Davenport, J.H., Siret, Y. and Tournier, (1988). Computer Algebra. *Academic Press, N.Y.*.

[di Primo, 1991] di Primo, B., (1991). Nichtstandard Erweiterungen von Differentialkörpern. *ETH-Thesis,* **9582**.

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

[Engeler, 1981B] Engeler, E. (1981). Algebras and Combinators. *Algebra Universalis*, 389-392.

[Engeler, 1988] Engeler, E. (1988). A Combinatory Representation of Varieties and Universal Classes. *Algebra Universalis,* **24**.

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

[Fehlmann, 1981] Fehlmann, T. (1981). Theorie und Anwendung des Graphmodells der kombinatorischen Logik. *Berichte des Instituts für Informatik der ETH* **41**.

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

[Kaucher, 1983] Kaucher, E. (1983): Solving Function Space Problems with Guaranteed Close Bounds. *In Kulisch, U. and Miranker, W.L.: A New Approach to Scientific Computation , Academic Press, New York, p 139-164.*

[Maeder, 1986] Mäder, R. (1986). Graph Algebras, Algebraic and Denotational Semantics. *ETH-Report* **86-04**.

[Pour-El & Richards, 1989] Pour-El, E., Richards, I.J. (1989). Computability in Analysis and Physics. *Springer*.

[Risch, 1969] Risch, R. (1969). The Problem of Integration in Finite Terms. *Transactions AMS, Vol. 139, p. 167-189.*

[Stetter, 1988] Stetter, H.J., (1988). Inclusion Algorithms with Functions as Data. *Computing, Suppl.6, p.213-224.*

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

[Weihrauch, 1980] Weihrauch, K. (1980). Rekursionstheorie und Komplexitätstheorie auf effektiven CPO-S. *Informatikberichte FernUniversität Hagen, 9/1980.*

[Weissinger, 1988] Weissinger, J. (1988). A Kind of Difference Methods for Enclosing Solutions of Ordinary Linear Boundary Value Problems. *Computing, Suppl. 6*, 23-32.

[Wolfram, 1988] Wolfram, S. (1988). Mathematica. *Addison-Wesley Publishing Company.*