Computer–Assisted Proofs in Analysis and Programming in Logic: A Case Study

Hans Koch[1]
Department of Mathematics, University of Texas at Austin
Austin, TX 78712

Alain Schenkel[2], Peter Wittwer[2]
Département de Physique Théorique, Université de Genève
Genève, CH 1211

Abstract. In this paper we present a computer–assisted proof of the existence of a solution for the Feigenbaum equation $\varphi(x) = \frac{1}{\lambda}\varphi(\varphi(\lambda x))$. There exist by now various such proofs in the literature. Although the one presented here is new, the main purpose of this paper is not to provide yet another version, but to give an easy–to–read and self contained introduction to the technique of computer–assisted proofs in analysis. Our proof is written in Prolog (**Pro**gramming in **log**ic), a programming language which we found to be well suited for this purpose. In this paper we also give an introduction to Prolog, so that even a reader without prior exposure to programming should be able to verify the correctness of the proof.

[1] Supported in Part by the National Science Foundation under Grant No. DMS–9103590, and by the Texas Advanced Research Program under Grant No. ARP–035.

[2] Supported in Part by the Swiss National Science Foundation.

Table of Contents

1. Introduction

The aim of this paper is to give a self contained introduction to the technique of computer–assisted proofs in analysis. We center our discussion around a proof of the existence of a function φ that satisfies, for some real number λ, the Feigenbaum equation

$$\varphi(x) = \frac{1}{\lambda}\varphi(\varphi(\lambda x)) \tag{1.1}$$

on the interval $|x| < \sqrt{2.7}$. Our solution $\varphi = \phi$ of this equation is even, has a quadratic maximum at $x = 0$, vanishes at $x = 1$, and extends analytically to the complex domain $|x^2 - 1| < 1.7$. The graph of this function is depicted in Fig.1.

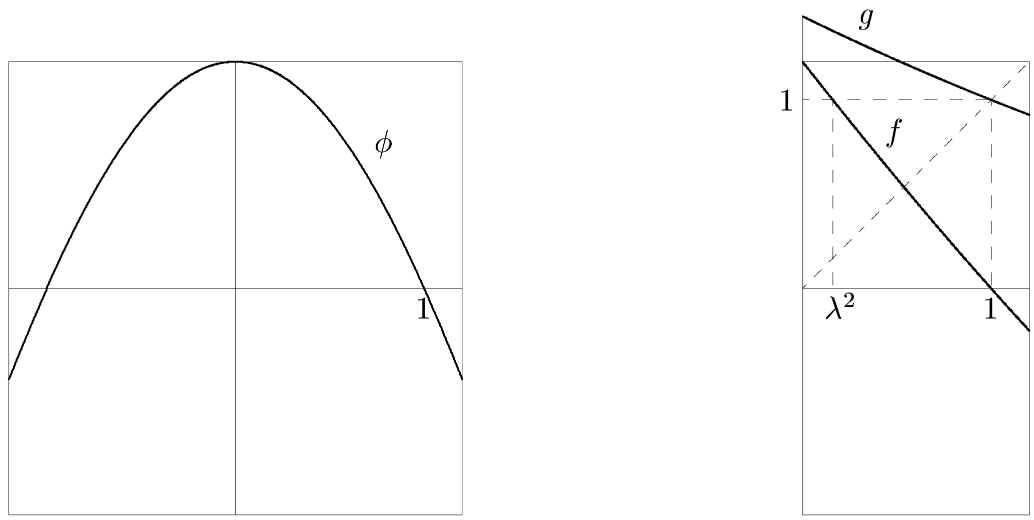

Fig. 1: Graphs of the function ϕ and of the corresponding functions f and g

We note that the existence of solutions to (1.1) is well known; see [La1$\cdots$ La4] for computer–assisted proofs and [E] for other methods and a review on the subject. The proof presented here is new, but as mentioned above, this is not the main point of this paper. For convenience later on, we shall now reformulate the Feigenbaum problem.

One of the basic properties of equation (1.1) is its scale invariance. That is, if φ is a solution, then $\frac{1}{s}\varphi(s\cdot)$ is again a solution, for any real number $s \neq 0$. The following normalization condition will be imposed to brake this scale invariance:

$$\varphi(1) = 0. \tag{1.2}$$

We note that by setting $x = 1$ in (1.1), we obtain from (1.2) the implicit equation $\varphi(\varphi(\lambda)) = 0$ for the scaling factor λ.

Considering now only even φ's, we can write $\varphi(x) = f(x^2)$ for some function f. In this case, equation (1.1) becomes equivalent to the pair of equations

$$f(z) = \frac{1}{\lambda}f(g(z)), \tag{1.3}$$

$$g(z) = f(\lambda^2 z)^2, \tag{1.4}$$

and the condition (1.2) becomes $f(1) = 0$. The graphs of the functions f and g which correspond to the solution ϕ are shown in Fig.1.

If $x = 1$ is the only zero of f in the domain of interest, then $f(1) = 0$ implies, together with equations (1.3) and (1.4), the additional normalization conditions $g(1) = 1$, $f(\lambda^2)^2 = 1$, and $g'(1) = \lambda$, where g' denotes the first derivative of g.

In what follows, we restrict our search for solutions of (1.3) and (1.4) to functions f and g which are analytic on the disk $|z-1| < 1.7$, and which satisfy all of the normalization conditions mentioned above. To this end, it is convenient to change coordinates and to choose $x = 1$ as the new origin. Define two functions F and G by setting $F(z) = f(1+z)$ and $G(z) = g(1+z) - 1$. Then the equations (1.3) and (1.4) can be rewritten as

$$F(z) = \frac{1}{\lambda} F(G(z)), \tag{1.5}$$

$$G(z) = F(-1 + \lambda^2 + \lambda^2 z)^2 - 1, \tag{1.6}$$

and the normalization conditions become

$$F(0) = 0, \quad F(-1+\lambda^2)^2 = 1, \quad G(0) = 0, \quad G'(0) = \lambda. \tag{1.7}$$

Equation (1.5) is known as the Schröder equation. It is of some interest on its own, since the solution F conjugates the function G to its linear part; formally $F(G(F^{-1}(z))) = \lambda z$.

Definition 1.1. *Let $\mathcal{B}$ be the Banach space of all analytic functions H, $H(z) = \sum_{i \geq 0} H_i \, z^i$, from the disk*

$$\Omega = \{ z \in \mathbb{C} |\; |z| < D \} \tag{1.8}$$

of radius

$$D = 1.7 \tag{1.9}$$

to the complex plane, which have real Taylor coefficients H_i and a finite norm

$$\|H\| = \sum_{i \geq 0} |H_i| \; D^i. \tag{1.10}$$

Furthermore, define $\mathcal{B}_0$ to be the subspace of all functions in $\mathcal{B}$ that vanish at $z = 0$.

In this paper we prove the following.

Theorem 1.2. *There exist functions F and G in $\mathcal{B}_0$, and a negative real number λ, such that for every $z \in \Omega$, the values $G(z)$ and $-1 + \lambda^2 + \lambda^2 z$ are in Ω, and the equations (1.5), (1.6), and (1.7) are satisfied.*

Our proof of Theorem 1.2 uses estimates that have been carried out by a computer, and that can be verified by other computers. The necessary computer program is included and will be discussed in detail in the following sections. We conclude this introduction with some remarks concerning our choice of programming language, etc. Other computer–assisted proofs and an extensive list of papers on interval methods can be found in the references [C, CC, EKW1⋯EKW2, EW1⋯EW2, FS, FL, KP, KW1⋯KW7, La1⋯La4, Lla, LLo, LR1⋯LR3, M, Ra, Se1⋯Se2, St] and [XSC], respectively.

Unfortunately, computer–assisted proofs are usually very hard to read. The computer programs tend to be lengthy, even for relatively simple problems, and it is often tedious to verify that mathematical expressions are correctly represented by their programmed versions. This is because the syntax of traditional programming languages like ADA, Basic, C, C++, Fortran, Modula2, Pascal etc. is quite different from the syntax that mathematicians are used to, and because in these programming languages, memory allocation, control structures and the like make up most of the programs. As a consequence, the simplicity of the ideas underlying computer–assisted proofs is often lost along the way.

There have been attempts to improve this situation, e.g. by adding syntactic features to a language like Pascal [EMO, XSC]. In this paper we propose to use the programming language Prolog (**Pro**gramming in **log**ic). Prolog allows for a syntax close to the one that mathematicians are used to. In addition, Prolog does not require writing code for memory allocation etc.

Most computer–assisted proofs in analysis rely on the contraction mapping principle for proving the existence of solutions to equations in Banach spaces, i.e., equations are written in the form of a fixed point problem for an appropriate operator $\mathcal{R}$. This requires, among other things, an estimate on the norm of the (Fréchet) derivative of $\mathcal{R}$. Traditionally, a considerable fraction of the program is taken up by the parts devoted to such estimates. With Prolog, however, it becomes natural to have the program generate these parts, by applying the chain rule of differentiation to the operator $\mathcal{R}$, which is usually composed of several "building blocks."

Prolog has been in use for many years now. In spite of that, it did not gain much popularity outside the Artificial Intelligence community, mainly because, until not too long ago, efficient implementations of the language were sparse and did not offer reasonable support for floating point arithmetic. More recently, good implementations which do support decent floating point arithmetic have become widely available, so that the interested reader should be able to find a Prolog system which can be used to verify our proof.

In this paper we use Prolog for all the advantages it offers for our type of application. We do not claim in any way that Prolog is the "perfect programming language." Indeed, Prolog is in many respects far from being perfect, and has, like any other language, its strong and its weak points. We do feel, however, that the main ideas which lead to the development of Prolog will play a role in the future of programming, and we would like to bring some of these ideas to the reader's attention.

This paper has been prepared by using the TEX text processing system. Those readers who have the corresponding .tex file can run it through the TEXcompiler. This will produce, along with the usual .dvi file, a file named proof.pl containing the extraction of the Prolog lines which make up the main part of our proof.

2. Prolog

At the time of this writing there exists no official standard for Prolog. Many of the better implementation, however, adhere to a *de facto* standard, known under the name of Edinburgh Prolog, which is described e.g. in [CM]. Most implementations also offer some additional features that go beyond the Edinburgh "standard". For reasons of compatibility, we chose not to use such features here, with one exception: we use floating point arithmetic, and we require that it be implemented as 64 bit arithmetic according to the IEEE standard, which will be described in Section 3.

There is one part of the Prolog syntax, the operator notation (see Subsection 2.2), that is not entirely standardized within Edinburgh Prolog. A consequence which is relevant here is that the so called precedence classes (certain positive integers) of predefined operators may vary from one implementation to the next. Thus, some readers may have to modify the program in order to get it to behave as intended on their Prolog systems. The necessary changes are minor and are described below.

We note that these issues can be ignored by the reader who is willing to believe that we actually ran our program once with positive outcome. In this case, it suffices to verify the correctness of the program in order to check the proof. Everything that is needed for this step — the program and a description of the Prolog syntax that was used to write it — is provided in this paper.

Our program was developed, for the most part, by using LPA–Prolog [LPA]. It was run with LPA–Prolog on a "80486" personal computer, and with SICStus Prolog [SP] on various Unix workstations. The running time was between 5 and 30 minutes, depending on the system used.

We continue this section with an introduction to Prolog. No additional knowledge of programming, beyond what is presented below, will be necessary to follow the rest of this paper. Should the reader prefer to learn about Prolog first from another source, the classic textbook [CM] is still a very good starting point. Possible continuations include the textbook [SS], or the book [R] which also gives a rather extensive list of Prolog implementations and references for further reading. In what follows, items related to Prolog will be highlighted using `this_special_font`.

2.1. The Syntax of Prolog Terms

In Prolog, a program consists of data, and there is a single data structure, the term. In order to define its syntax, we introduce the notion of atoms, variables, and numbers:

i) An atom is one of the following four: (1) a finite sequence of alphanumeric or underscore characters, i.e., characters chosen from

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 0 _
```

starting with a lower case letter; (2) any finite sequence of characters chosen from

```
+  -  *  /  .   ~  ?   :   <  >  =  ^ @  #  $  &
```

(3) the exclamation mark, the semicolon, or the pair of square brackets, i.e.,

`!` `;` `[]`

(4) an "arbitrary" sequence of "arbitrary" characters in single quotes. We shall not specify here the meanings of "arbitrary." In what follows, the only (important) instance of this case will be the atom `','`.

ii) A variable is a finite sequence of alphanumeric or underscore characters, starting with an upper case letter or an underscore. Variables starting with an underscore are called anonymous.

iii) A number is any sufficiently small nonnegative integer, written in base ten "everyday notation," using numeric characters. The meaning of "sufficiently small" depends on the Prolog implementation and typically stands for "smaller than 2^{31}."

As mentioned in the introduction, our proof requires a Prolog implementation that provides 64bit IEEE floating point arithmetic. Most Prologs that provide such numbers also allow to use "them" as syntactic elements. Since we have no need for using such numbers on a syntactic level, we stick, as far as the syntax is concerned, with the Edinburgh definition.

Prolog terms are now defined inductively as follows. A Prolog term comprises a functor and zero, one, or more arguments. The arguments, if there are any, can be either variables, numbers, or again Prolog terms. A functor is characterized by its name (an atom) and its arity (the number of arguments). We will refer to a functor with name `funtor_name` and arity `N` by using the customary symbol `functor_name/N`.

The standard syntax for a prolog term with functor `functor_name/N`, in the cases $\texttt{N} = 0, 1, 2, \cdots$ is (the etc.-dots $\cdots$ are not to be confused with the Prolog atom `...`)

```
functor_name
functor_name(First_argument)
functor_name(First_argument, Second_argument)
functor_name(First_argument, Second_argument, ···, Last_argument)
```

Here, `functor_name` stands for any legal Prolog atom. Arguments have to be separated by commas, but space characters can be added before and after each argument to enhance readability. No white space is allowed between the atom `functor_name` and the opening bracket.

2.2. The Operator Syntax for Prolog Terms

One fact about Prolog that is of practical importance is that it provides an alternative syntax for terms with one or two arguments: such terms can be written in operator notation. The functor becomes an operator, and, depending on whether this operator is declared as postfix (type `xf` or `yf`), prefix (type `fx` or `fy`), or infix (type `xfx`, `yfx`, `xfy`, or `yfy`), its name is written after the argument, before the argument, or between the first and the second argument.

Many operators are already known to standard Prolog. The following is a complete list of all those predeclared operators which are used in this paper. Each entry has the form of a declaration `op(P,Type,Name)`, specifying the precedence class, the type, and the name of the operator.

```
op(1200,fx,:-)
op(1200,xfx,:-)
op(1000,xfy,',')
op( 700,xfx,=)
op( 700,xfx,=..)
op( 700,xfx,is)
op( 700,xfx,<)
op( 700,xfx,>)
op( 700,xfx,=<)
op( 700,xfx,>=)
op( 500,yfx,+)
op( 500,yfx,-)
op( 500,fx ,-)
op( 400,yfx,*)
op( 400,yfx,/)
```

The precedence classes specify how a term like

```
2+3*4
```

has to be interpreted. According to the declarations listed above, it has to be read as `+(2,*(3,4))`. The type `yfx` indicates that the operator is left associative, which means e.g. that

```
2/3*4
```

is equivalent to `*(/(2,3),4)` and not to `/(2,*(3,4))`. The operator notation of the last term is

```
2/ (3*4)
```

i.e., we can always use parentheses to overrule the given associativity rules or operator precedence. The space between the operator name `/` and the opening parentheses is necessary here (in contrast to the standard syntax for terms) in order to avoid possible ambiguities. Thus, we shall always put such spaces, even though most Prolog compilers do not insist on that. Parentheses can also be used redundantly (as we often do in our program) to encapsulate arguments for better readability, e.g., `2/3*4` may be written as `(2/3)*4`.

More generally, a letter `x` or `y` in the `Type` of an operator `given/N` specifies what kind of argument can be used at the corresponding position in a term with this operator. Restrictions apply (only) if the argument is also a term written in operator notation, without enclosing parentheses. In this case, an `x` indicates that the precedence class of the argument–operator has to be lower (a smaller integer) than that of `given/N`. A `y` is less restrictive and allows the two precedence classes to be equal as well.

We note that the only purpose of precedence classes is to define an order among (classes of) operators, i.e., the absolute size of these numbers is of no importance. For this reason, the precedence class values of predefined operators vary, unfortunately, from one implementation of Prolog to another (though usually not their ordering). Thus, the reader who wishes to run our program should compare the precedence class of predefined operators with the ones listed above . In case the values do not all agree, some of the lines in the program part presented in Subsection 2.8 below may have to be adapted accordingly.

We shall not present the syntax rules for general expressions involving operators in any more detail. The reader is warned, however, that without such specifications (which are missing in most dialects of Prolog) general expressions involving operators can be ambiguous. We avoid this problem by using only operator expressions which can be translated to the original Prolog syntax by applying just the associativity rules and precedence class rules explained above.

2.3. The List Notation

Prolog offers, besides the operator notation, additional "syntactic sugar" for certain terms with the functor `./2`. Namely, terms of the form

```
.(Term, Again)
```

where `Term` is an arbitrary Prolog term, and where `Again` is either the atom `[]` or of the form `.(Term, Again)`, can be written in list notation. For example

```
[1,2,3]
```

is just a simpler way of writing

```
.(1, .(2, .(3,[])))
```

and general lists are defined analogously. In this context the atom `[]` is called the empty list. Another piece of notation, which defines `[X|T]` to be the same as `.(X,T)`, provides a convenient way of adding a term at the beginning of a list. In particular, if `T` stands for the list `[Y,Z]` then `[X|T]` is identical to `[X,Y,Z]`. There is an analogous way of prepending two or more terms to a list. For example, `[X,Y|T]` is defined as `.(X,.(Y,T))`, which implies e.g. that `[1,2|[3,4]]` is the same as `[1,2,3,4]`. We note that `[a]` is equivalent to `[a|[]]`, but not to `[a,[]]`. The latter is a list with two elements; it is syntactically equivalent to `.(a, .([],[]))`.

2.4. The Syntax of Prolog Programs

Syntactically, a Prolog program is a sequence of Prolog terms, each followed by a full stop and a line feed character. (Most versions of Prolog will allow "layout" characters different from line feed.) Such a term in a program is called a clause. If the functor of a clause is the infix operator `:-/2`, then we call the clause a non unit clause. Otherwise, it will be called a unit clause, unless its functor is the prefix operator `:-/1`, in which case it is called a headless clause. There are restrictions (which have been partly eliminated in some newer Prolog dialects) on the type of terms that can be used as clauses. In particular, the first argument of `:-/2` in a non unit clause has to be a term, i.e., it cannot be a variable or a number, and the functor of this term is not allowed to be one of the operators `:-/1`, `:-/2`, and `','/2`. The following is a Prolog program:

```
head.
head :- body.
```

It consists of a unit clause and a non unit clause. The first argument of `:-/2` in a non unit clause is called the head of the clause, and the second argument is called the body. Unit clauses have a head only.

Given a program, we define a predicate to be the set of all clauses whose heads have the same functor, and we (mis)use this functor to refer to the corresponding predicate. Thus, the above example program defines one predicate: `head/0`.

Below we will see that the operator `','/2` has a special meaning in non unit clauses of the form `A :- B0 ',' B1 ',' ···` and in headless clauses of the form `:- B0 ',' B1 ',' ···`. For better readability, Prolog allows for the quotes to be suppressed in these cases, i.e., one can simply write `A :- B0,B1,···` and `:- B0,B1,···` .

2.5. Matching Terms

Assume that `T1` is some Prolog term that does not contain any variables. A term `T2` is said to match `T1` if it is the same term, or if it contains variables that can be chosen (by instantiating them to numbers or to Prolog terms not containing variables) in such a way that it becomes the same term.

According to this rule, the term `i(1,2)` is matched by `i(1,2)` and `i(X,Y)`, but not by `i(0,1)`. The term `i(X,X)` does not match the term `i(1,2)`, since there is no choice of the variable `X` which makes the two terms identical. On the other hand, the term `i(_,_)` does match the term `i(1,2)`, since every anonymous variable is defined to be independent of any other variable.

The matching relation is symmetric. That is, if `T2` matches `T1` then `T1` matches `T2`. In general, matching is defined recursively as follows:

i) If `X` is an uninstantiated variable and `T` is instantiated to a term or a (representable) number, then `X` matches `T` and `X` becomes instantiated to whatever `T` is. If `X` and `Y` are both uninstantiated variables then `X` matches `Y`, and `X` and `Y` become synonyms for the same (uninstantiated) variable. An anonymous variable matches anything (that is syntactically correct).

ii) Two Prolog terms match if they have the same functor name and the same number of arguments, and if corresponding arguments match.

For example, the term `f(X,i(1,Y))` matches the term `f(Y,i(X,1))` but not `f(Y,i(X,2))`. Also, according to the above definition, `X` matches `f(X)`, but `i(1,X)` does not match `i(X,f(X))`.

2.6. Program Execution

In this section we give an informal account of how a Prolog program is "executed." As explained earlier, the intention is not to give a complete description of Prolog, but only to provide enough information so that the reader can check the correctness of our program.

Prolog, as a programming system, is not just the language whose syntax we have introduced above, but it is this language together with an "inference engine" that does some sort of reasoning, as will now be described. There is a way of making Prolog look at a file (in our case the file proof.pl) that contains a program. How this is done precisely is of no importance for the moment. It suffices to know that Prolog starts to look at such a file from the top. In particular, since the various parts of our program will be presented in the order they appear in the file proof.pl, we can imagine Prolog to be watching over our shoulders and reading along, as we go over each of these parts. If Prolog encounters a unit clause or a non unit clause, it does nothing except "memorizing" the clause by adding it to the bottom of some (internal) list, also called the database.

The idea is that Prolog acquires knowledge by adding clauses to its database. Roughly speaking, a unit clause `H` declares a fact "`H` is true", and a non unit clause `H:-B` is a rule "`H` is true if `B` is true" that can be used to infer new facts from facts that are already known. Such an inference is started when Prolog encounters a headless clause.

A headless clause `:-G` is interpreted as an instruction to "satisfy `G`". In this context, `G` is also called a goal. When Prolog encounters such a headless clause, it tries to satisfy `G` (in a way to be explained) by using the clauses that are already in the database. If this is impossible, Prolog stops and communicates to the user some sort of error message. Otherwise, Prolog continues reading the program until it reaches the next headless clause, or until it arrives at the end of the file containing the program, in which case it also stops and communicates to the user that it has been successfully building up a database. (If required one can then ask Prolog to look at another file containing more clauses, etc.)

We now explain how Prolog tries to satisfy a goal `G`. First, we note that there are two possible outcomes: either `G` is satisfied, or else it fails. A satisfied goal is interpreted as a true statement. By contrast, a failed goal should not be interpreted as a false statement, but rather as a statement whose truth value is not determined by the facts and rules contained in the database.

Assume now that `G` is a goal whose functor is not `','/2`. In its attempt to satisfy this goal, Prolog starts at the beginning of the database and checks, one clause after another, whether or not `G` matches the head of the clause (in the sense explained in Subjection 2.5). If Prolog finds a unit clause `H` that matches `G`, then the goal `G` is satisfied. Alternatively, if Prolog gets to a non unit clause `H:-B` whose head `H` matches `G`, then it tries to satisfy the goal `B` (possibly with some of its variables being instantiated through the matching with `H`), and if `B` becomes true then so does `G`. To satisfy `B`, Prolog proceeds recursively. In particular, if the functor of `B` is not `','/2`, then the database is searched (again from the top) for a clause whose head matches `B`, etc. If `B` fails, then Prolog backtracks and tries to satisfy `G` in a different way, by continuing its search of the database with the clauses following `H:-B`, in the same state as it would have been in the case where `G` did not match `H`. Should Prolog arrive at the end of the database, with all clauses checked and none of them satisfying `G`, then the goal `G` fails.

We note that by matching the goal `G` with the head of a clause `C` in the database, Prolog makes a choice: `C` becomes the first clause used in the current attempt to prove (satisfy) `G`. Before making this choice, Prolog saves all the information needed to be able to backtrack to the same pre–choice state. Prolog will backtrack to this state later if no proof of `G` can be found that starts with `C`, or if such a proof was found but an alternative proof is needed. In both of these cases, Prolog tries to satisfy `G` in a different way, by continuing its search of the database with the clauses following `C`. From this point on, the above–mentioned pre–choice state is no longer available for backtracking, since both choices (using `C`, or skipping `C`) have already been tried.

Consider now the case where the goal is of the form `G1','G2`. The operator `','/2` corresponds to the logical "and". But as a goal, `G1','G2` should be read as "`G1` and then `G2`". Thus, in order to satisfy `G1','G2`, Prolog has to satisfy first `G1` and then `G2`. Assume now that `G1` was found to be true. The next step is to satisfy the goal `G2`. Here, it is important to note that some variables in `G2` may be instantiated as a result of matches from previous steps. Thus, if `G2` fails at this point, then Prolog will consider alternative ways of satisfying `G1`, by backtracking to the most recently saved state available and proceeding as described above. Each time Prolog finds a new proof for `G1`, it tries again to satisfy `G2`, and if this succeeds then `G1','G2` is true. The goal `G1','G2` will fail only if `G1` fails, or if `G2` fails after each proof of `G1`.

2.7. Built–in Predicates.

Besides predeclared operators, Prolog also includes a set of predefined predicates. The following is a complete list of all built–in predicated that are used in our program:

```
=/2
=../2
is/2
>/2
</2
=</2
>=/2
asserta/1
op/3
write/1
nl/0
!/0
```

We note that the list in Subsection 2.2 contains some additional operators that are not mentioned here. Those operators have a special meaning only on a syntactic level.

Some built–in predicates are provided for convenience only (`=/2` is of this type), i.e., they could also be defined by the user. The others are not predicates in the sense defined in Subsection 2.4. Some of them behave just like ordinary predicates, from the point of view of program execution, but include features that the user could not implement (`=../2` is of this type), or not implement easily (`is/2`, `</2`, `>/2`, `=</2` and `>=/2` are of this type). Still others (like asserta/1, op/3, write/1, nl/0 and !/0) make goals that are always satisfied, but that have side–effects. We now explain the semantics of each of these predicates.

=/2 behaves as if defined as X=X, i.e., it simply matches two terms.

=../2 allows one to decompose a given term T into its functor and its arguments. If L is an uninstantiated variable and T is instantiated to functor_name(Arg1, Arg2, ⋯), then T =..L is satisfied and L gets instantiated to the Prolog list [functor_name, Arg1, Arg2, ⋯].

is/2 evaluates arithmetic expressions. If X is an uninstantiated variable, and if T is a number (or representable number, see Section 3) or a Prolog term whose functor is one of the basic arithmetic operators (unary minus, addition, subtraction, multiplication, and division) and whose arguments are (variables instantiated to) either numbers or again terms of the same form, then X is T is satisfied, and X gets instantiated to the (approximate, in the sense of Section 3) evaluation of the arithmetic expression T.

X > Y is satisfied if the variables X and Y are instantiated to numbers (any representable number in the sense of Section 3 is allowed here), and if X is greater than Y. The situation is analogous for goals with the predicates </2, >=/2 and =</2, which implement the order relations "less than", "greater than or equal", and "less than or equal", respectively.

asserta(T) is satisfied if T is a Prolog term that can be interpreted as a clause. This clause is then added at the beginning of the Prolog database.

op(P,Type,Name) is satisfied if P is a positive integer less than the precedence class of :-/1 and :-/2, if Type is one of the atoms xf, yf, fx, fy, xfx, xfy, yfx or yfy, and if Name is any atom (except for names of predefined operators). As a side–effect, the Prolog syntax is changed: the atom Name is considered to be the name of an operator with precedence class P and of associativity type Type, as explained in Subsection 2.2.

write(A) is satisfied if A is an atom. As a side–effect, A is written to the current output stream (e.g. the terminal).

nl is always satisfied, and it has the side–effect of writing a newline character to the current output stream.

! (read "cut") always succeeds, and as a side–effect, it commits Prolog to all the choices made since the parent goal was unified with the head of the clause containing the cut. Our programs uses cuts only to help the Prolog system economize memory usage; it runs correctly even if all cuts are removed. Thus, when reading our program, the reader can safely ignore the cuts. (See [CM, R, SS] for an introduction to the cut predicate and [Llo] for a possibility of defining its semantics.)

2.8. Syntax Used in the Proof

Any line that is highlighted in `this_special_font` and appears in "display style", i.e.

```
like_this
```

is part of our program from now on. To begin with, we declare a few operators which will allow us to write the rest of the program in a syntax that is close to the familiar notation of mathematics:

```
:-op(700,xfx,is_a_safe_upper_bound_on).
:-op(700,xfx,is_a_safe_lower_bound_on).
:-op(700,xfx,enlarges).
:-op(700,xfx,contains).
:-op(700,xfx,approximates).
:-op(700,xfx,lt).
:-op(600,xfy,of).
:-op(500,xfx,at).
:-op(300,yfx,o).
:-op(200,xf,inverse).
:-op(200,xfx,**).
```

Later on, we will define predicates associated with the first six operators declared in this list. The operators `at/2`, `of/2`, `o/2`, `inverse/1`, and `**/2`, are used for syntactic convenience only.

3. 64bit IEEE Arithmetic and Rounding

The 64bit IEEE standard for floating point arithmetic [IEEE] specifies two things: a format for floating point numbers, and rules concerning rounding after the operations $+$, $-$, $*$ and $/$.

Unfortunately, the IEEE standard comes in several flavors. In the present context, this means that the semantics of the predicate `is/2` may differ from one Prolog system to the next, even if both comply with "the" IEEE standard. In our program we avoid this problem by restricting `is/2` to a domain where all versions of the standard agree.

In 64bit IEEE arithmetic, a number is represented with 64 bits of memory: 1 bit for the sign of the number, 11 bits for an exponent, and 52 bits for a mantissa. There is a convention which identifies the state of a bit with the integers zero and one, e.g., any given state of the 11 exponent bits determines, via the base 2 representation for nonnegative integers, an integer e in the range $0 \leq e \leq 2047$. A real number r is now associated with any bit pattern of 64 bits for which $0 < e < 2047$, and r is defined by the equation

$$r = \pm 1.m \; 2^{e-(2^{10}-1)}, \tag{3.1}$$

where $+$ or $-$ is chosen according to whether the sign bit is zero or one, and where m is a base 2 mantissa given by the sequence of zeros and ones associated with the 52 mantissa bits. In addition, the bit pattern consisting of 64 zeros is used to represent $r = 0$.

Definition 3.1. *A real number r is called representable if it is either zero, or if it can be written in the form (3.1), subject to the above–mentioned restrictions on e and m. The set of all representable numbers will be denoted by $\mathcal{R}$.*

Remark. On many systems, integers between -2^{31} and $2^{31}-1$ are represented in the "32bit two's complement format", and not in the 64bit IEEE floating point format. However, since all these integers are representable numbers, too, and since 64bit IEEE arithmetic computes without rounding errors with these integers, there will be no need for distinguishing the two internal representations, and we will not make any mention of it anymore.

We note that the smallest and largest positive representable numbers are $2^{-1022} \approx 2.2 \cdot 10^{-308}$ and $(2-2^{-52}) \cdot 2^{1023} = (1-2^{-53}) \cdot 2^{1024} \approx 1.8 \cdot 10^{308}$, respectively. Other "interesting" numbers are the smallest representable number larger than one, $1+2^{-52}$, and the next nearest neighbor in $\mathcal{R}$ to the left of one, $1-2^{-52}$.

In what follows, a number described by the 64bit IEEE standard will be referred to as "IEEE number". This includes not only what we call here representable numbers, but (among others) also some additional numbers between $\pm 2^{-1022}$, associated with bit patterns for which $e = 0$.

Besides defining a number format, the IEEE standard also specifies the precision to which arithmetic computations have to be carried out. Let r_1 and r_2 be two arbitrary IEEE numbers, and let $\#$ be any of $+$, $-$, $*$, or $/$. If $r_1 \# r_2$ is not an IEEE number, but not larger or smaller than every IEEE number (case of an overflow), then the approximate result computed for $r_1 \# r_2$ is guaranteed to be either the largest IEEE number smaller than $r_1 \# r_2$, or the smallest IEEE number larger than $r_1 \# r_2$. If $r_1 \# r_2$ is an IEEE number, then the IEEE standard prescribes that the computed result for $r_1 \# r_2$ be exact. In particular, the computed result is exact if either $r_1 = 0$, or if $r_2 = 0$ and $\#$ is not division. Also, if we avoid overflow or underflow (see below), then multiplication and division of a representable number by two are always carried out exactly, since, in terms of the representation (3.1), this just corresponds to increasing or decreasing e by one. The IEEE standard also requires that the tests $r_1 = r_2$, $r_1 < r_2$, $r_1 > r_2$, $r_1 \leq r_2$, and $r_1 \geq r_2$ be implemented correctly.

One of the problems of floating point arithmetic, besides overflow, is the so called "silent underflow to zero." For example, if r is a sufficiently small positive IEEE number, then a computer conforming with the IEEE standard can approximate $r * r$ by zero. This behavior is usually not considered a problem, and typically no execution error is raised (whence the expression silent underflow). But there are cases where it is important to know whether the result of an arithmetic operation is really zero or just very close to zero. Below, we will restrict our usage of IEEE numbers to a subset for which neither underflow nor overflow can occur.

For convenience later on, we define now two predicates, `negative_power_of_two/2` and `positive_power_of_two/2`, for computing integer powers of two within the representable range. We start with the case of negative powers. The desired effect is that if `N` is a given integer, then `negative_power_of_two(X,N)` becomes true as a goal if and only if $0 \leq \mathtt{N} \leq 1022$ and $\mathtt{X} = 2^{-\mathtt{N}}$. Our definition of `negative_power_of_two/2` starts with the true statement $\mathtt{1} = 2^{\mathtt{0}}$,

```
negative_power_of_two(1,0):-
 !.
```

The other cases are covered recursively by the clause

```
negative_power_of_two(X,N):-
 N>0,
 N=<1022,
 M is N-1,
 negative_power_of_two(Y,M),
 X is Y/2,
 !.
```

where we first check that `N` is within the allowed range, then compute $2^{\texttt{-(N-1)}}$, and then divide the result by two. The predicate `positive_powers_of_two` can now be defined by using the fact that $2^{\texttt{N}} = 1/2^{\texttt{-N}}$,

```
positive_power_of_two(X,N):-
 negative_power_of_two(Y,N),
 X is 1/Y,
 !.
```

In order to avoid any kind of difficulties related to underflow and overflow, and to make our program compatible with every version of the 64bit IEEE standard, we will check explicitly that arithmetic operations are carried out within a (somewhat arbitrary) "safe range".

Definition 3.2. *We define the safe range $\mathcal{S}$ to be the set*

$$\mathcal{S} = \left\{r \in \mathcal{R} \middle|\ 2^{-500} < |r| < 2^{500}\right\} \cup \{0\}, \tag{3.2}$$

where $\mathcal{R}$ is the set of representable numbers defined above.

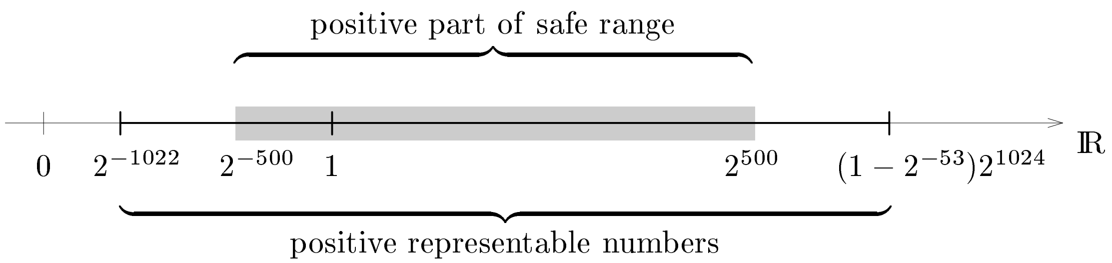

Fig. 2: Positive representable numbers and positive part of safe range

In Prolog we define the positive (part of the) safe range with a predicate that makes `lower_bound_on_positive_safe_range(X)` true if and only if `X` is 2^{-500}, and a predicate that makes `upper_bound_on_positive_safe_range(X)` true if and only if `X` is 2^{500}. In order to avoid having to compute $2^{\pm 500}$ each time one of these predicates is used, we will define them in the form `:-B,asserta(H)` instead of `H:-B`. This is done as follows:

```
:- negative_power_of_two(X,500),
   asserta(lower_bound_on_positive_safe_range(X)).
```

We note that this is the first place where Prolog encounters a headless clause. Prolog starts by trying to satisfy the goal `negative_power_of_two(X,500)`. This succeeds, while `X` is being instantiated to 2^{-500}. Then the clause `lower_bound_on_positive_safe_range(X)`, where `X` is now equal to 2^{-500}, is prepended to the internal database. This defines a predicate `lower_bound_on_positive_safe_range/1`. Similarly,

```
:- positive_power_of_two(X,500),
   asserta(upper_bound_on_positive_safe_range(X)).
```

In what follows, r_1 and r_2 denote fixed but arbitrary numbers in the safe range, except in a division r_1/r_2, where we exclude $r_2 = 0$. In addition, a "computation" will always refer to a computation carried out by a given but arbitrary system that conforms to the 64bit IEEE standard. These restrictions guarantee e.g. that no underflow or overflow occurs in the computation of $r_1 \# r_2$, and that the result always has the correct sign (plus, minus, or zero).

Assume now that $r_1 \# r_2$ is nonzero. Then the computation of $r = r_1 \# r_2$ yields a nonzero representable number r_c which is not necessarily in the safe range, but which can still be multiplied by $u = 1 + 2^{-52}$ and $d = 1 - 2^{-52}$, without causing underflow or overflow. Let $r_c^>$ and $r_c^<$ be the representable numbers obtained from a computation of the product $r^> = r_c * x$ and $r^< = r_c * y$, respectively, where $(x, y) = (u, d)$ if r_c is positive, and $(x, y) = (d, u)$ if r_c is negative. By using the IEEE specifications given above, it is easy to verify that $r_c^<$ is either the nearest or next nearest neighbor in $\mathcal{R}$ to the left of r_c, and that $r_c^>$ is either the nearest or next nearest neighbor in $\mathcal{R}$ to the right of r_c. As a result, we have $r_c^< < r < r_c^>$.

Definition 3.3. *Given a representable number x, we say that a representable number t is a safe upper bound on x if either $t = x = 0$, or if there exists a number $s \in \mathcal{S}$, with the same sign as x, such that $x < t \leq s$; and we say that t is a safe lower bound on x if either $t = x = 0$, or if there exists a number $s \in \mathcal{S}$, with the same sign as x, such that $s \leq t < x$.*

Below, whenever we compute $r_c^<$ and $r_c^>$, we will also verify that these numbers are safe lower and upper bounds, respectively, on the number r_c. These two properties together imply that both $r_c^<$ and $r_c^>$ are in the safe range.

This shows that (and how) it is possible to compute bounds on the result of an arithmetic operation using IEEE floating point arithmetic. In order to implement these ideas in Prolog, we first define two predicates `up/1` and `down/1`, such that `up(Up)` is true for `Up`$= 1 + 2^{-52}$,

```
:- negative_power_of_two(X,52),
   Up is 1+X,
   asserta(up(Up)).
```

and `down(Down)` is true with `Down`$= 1 - 2^{-52}$,

```
:- negative_power_of_two(X,52),
```

```
   Down is 1-X,
   asserta(down(Down)).
```

Let now r_c = `Y1` be a nonzero representable number obtained from a computation of $r = r_1 \# r_2$ with r_1 and r_2 in the safe range. Then the upper bound $r_c^{>}$ on r, which was described above, can be computed by using the following predicate which makes `Y2 is_a_safe_upper_bound_on Y1` true if `Y2` is equal to $r_c^{>}$ and a safe upper bound on `Y1`. The case of positive `Y1` is covered by the clause

```
Y2 is_a_safe_upper_bound_on Y1:-
 Y1>0,
 up(Up),
 Y2 is Up*Y1,
 upper_bound_on_positive_safe_range(U),
 Y2<U,
 !.
```

and the following applies if `Y1` is negative:

```
Y2 is_a_safe_upper_bound_on Y1:-
 Y1<0,
 down(Down),
 Y2 is Down*Y1,
 lower_bound_on_positive_safe_range(L),
 Lm is -L,
 Y2<Lm,
 !.
```

According to the Definition 3.2 we also have to add a clause like

```
Z is_a_safe_upper_bound_on Z:-
 Z>=0,
 Z=<0,
 !.
```

Similarly, the predicate `is_a_safe_lower_bound_on/2` is used to compute the number $r_c^{<}$ and to perform the indicated check. It can be expressed easily in terms of the predicate `is_a_safe_upper_bound_on/2` as follows:

```
X2 is_a_safe_lower_bound_on X1:-
 Y1 is -X1,
 Y2 is_a_safe_upper_bound_on Y1,
 X2 is -Y2,
 !.
```

4. Interval Analysis

In this section we will describe in general terms the type of bounds that are used in computer–assisted proofs. The description is independent of the other parts of the paper, but not vice versa: we will refer to this section for the definition of a "bound on a map" and for the notion of "standard sets".

Given any set Σ, denote by $\mathcal{P}(\Sigma)$ the set of all subsets of Σ. Let now Σ and Σ' be two sets, and let f and g be maps from $\mathcal{D}_f \subseteq \mathcal{P}(\Sigma)$ and $\mathcal{D}_g \subseteq \mathcal{P}(\Sigma)$, respectively, to $\mathcal{P}(\Sigma')$.

Definition 4.1. *We say that g is a bound on f, in symbols $g \geq f$, if $\mathcal{D}_f \supseteq \mathcal{D}_g$, and if $f(S) \subseteq g(S)$ for all $S \in \mathcal{D}_g$.*

Bounds of this type have some general properties which make it possible to estimate complicated maps in terms of simpler ones, and to "mechanize" such estimates. In particular, if $g = g_1 \circ g_2$ (composition) is well defined, with $g_1 \geq f_1$ and $g_2 \geq f_2$, then $f = f_1 \circ f_2$ is well defined and g is a bound on f.

In order to implement this idea in practice, we need to be able to verify that bounds can be composed, i.e., that the range of one bound is contained in the domain of the next. With this in mind we adopt the following procedure:

a) Given Σ and Σ', we specify two collections of sets,

$$\mathrm{std}(\Sigma) \subset \mathcal{P}(\Sigma), \quad \mathrm{std}(\Sigma') \subset \mathcal{P}(\Sigma'). \tag{4.1}$$

b) Given a map

$$f\colon \mathcal{P}(\Sigma) \supseteq \mathcal{D}_f \to \mathcal{P}(\Sigma')\,, \tag{4.2}$$

we construct a bound on f within the class of maps

$$g\colon \mathrm{std}(\Sigma) \supseteq \mathcal{D}_g \to \mathrm{std}(\Sigma')\,. \tag{4.3}$$

The elements of $\mathrm{std}(\Sigma)$ will be referred to as the standard sets for Σ. Unless specified otherwise, the standard sets for a Cartesian product $\Sigma \times \Sigma'$ will be defined by setting

$$\mathrm{std}(\Sigma \times \Sigma') = \mathrm{std}(\Sigma) \times \mathrm{std}(\Sigma'). \tag{4.4}$$

In most of our applications, the primary object of interest is not a map on power sets, but a map φ from some subset $\mathcal{D}_\varphi$ of Σ to Σ'. In this case, the indicated procedure is applied after we lift φ in the canonical way to a set map f with domain $\mathcal{D}_f = \mathcal{P}(\mathcal{D}_\varphi)$, by setting

$$f(S) = \{s' \in \Sigma' \mid s' = \varphi(s) \text{ for some } s \in S\} \tag{4.5}$$

for every $S \in \mathcal{D}_f$. By definition, a map g is a bound on f if and only if $S \in \mathcal{D}_f$ and $f(S) \subseteq g(S)$, for all $S \in \mathcal{D}_g$. Here, the latter holds if and only if $s \in \mathcal{D}_\varphi$ and $\varphi(s) \in g(S)$, whenever $s \in S \in \mathcal{D}_g$.

In this paper, a bound g on $f\colon \mathcal{P}(\Sigma) \supseteq \mathcal{D}_f \to \mathcal{P}(\Sigma')$ will be specified in Prolog by means of clauses for a predicate `contains/2`. The definition will be such that if `S` $\in \mathrm{std}(\Sigma)$, then `G contains f(S)` is true if and only if `S` is in the domain of g and `G`$= g($`S`$)$.

5. Elementary Operations with Real Numbers

In this section we introduce a systematic way of computing "with real numbers". Following the procedure outlined in Section 4, we start by defining the standard sets for $\mathbb{R}$.

Definition 5.1. std($\mathbb{R}$) *is defined to be the collection of all closed real intervals* `i(X,Y)` *of the form*

$$\mathtt{i(X,Y)} = \{r \in \mathbb{R} \mid \mathtt{X} \le r \le \mathtt{Y}\}, \tag{5.1}$$

with $\mathtt{X} \le \mathtt{Y}$ *elements of* $\mathcal{S}$.

Consider now the function unary minus which maps a real number $r \in \mathbb{R}$ to its negative $-r$. The canonical lift of this map to $\mathcal{P}(\mathbb{R})$ associates to any given $I \in \mathcal{P}(\mathbb{R})$ the set

$$-(I) = \{r \in \mathbb{R} \mid r = -x \text{ for some } x \in I\}. \tag{5.2}$$

A bound g, in the sense of Section 4, on the lifted unary minus is obtained by setting $\mathcal{D}_g = \mathrm{std}(\mathbb{R})$ and defining $g(I) = -(I)$ for all sets $I \in \mathcal{D}_g$. The following clause for `contains/2` implements this definition, i.e., for any given standard set `I`, the goal `I1 contains -I` becomes true if and only if $\mathtt{I1} = g(\mathtt{I})$.

```
i(X1,Y1) contains -i(X,Y):-
 X1 is -Y,
 Y1 is -X,
 !.
```

Here, we have used that if r is a safe number then so is $-r$, and that the computation of $-r$ from r is done exactly.

Our second bound is on the absolute value function. Let `i(P,Q)` be a fixed but arbitrary standard set for $\mathbb{R}$. It is easy to check that the following holds independently of whether this set contains zero, or only positive numbers, or only negative numbers: if `X` is the largest of the numbers zero, `P` and `-Q`, and if `Y` is the negative of the smallest of the same three numbers, then `i(X,Y)` is a standard set that contains the absolute value $|r|$ of any number r in `i(P,Q)`. In Prolog, this fact is stated by the clause

```
i(X,Y) contains abs(i(P,Q)):-
 N is -Q,
 sort_numbers([0,P,N],[R,_,X]),
 Y is -R,
 !.
```

where `0` is the number zero. The predicate `sort_numbers/2` used here is defined in Subsection 13.3. If `L1` is a list of numbers, then `sort_numbers(L1,L2)` is satisfied if and only if `L2` matches the list which is obtained by sorting the members of `L1` in increasing order. As in the case of the unary minus, the given clause for `contains/2` defines a bound with domain std($\mathbb{R}$) on the (set map associated with the) function considered.

The bounds to be defined next deal with functions of the form $\varphi((r_1, r_2)) = r_1 \# r_2$, where $\#$ is one of the binary operators $+$, $*$, or $/$. One of the things that distinguishes these functions from the ones discussed so far is that they need not be evaluated exactly on the computer, even if r_1 and r_2 are in $\mathcal{S}$, as we shall now assume. However, as far as bounds are concerned, it is sufficient to find an interval i(X1,Y1) that contains $r_1\#r_2$, given an interval i(X,Y) that contains the computed value for $r_1\#r_2$. The following predicate enlarges/2 serves this purpose:

```
i(X1,Y1) enlarges i(X,Y):-
 X1 is_a_safe_lower_bound_on X,
 Y1 is_a_safe_upper_bound_on Y,
 !.
```

To be more precise, if X≤Y are given representable numbers (not necessarily in the safe range), then i(X1,Y1) enlarges i(X,Y) is satisfied if and only if the indicated safe lower bound X1 and safe upper bound Y1 can be found, in which case i(X1,Y1) is a standard set with the above–mentioned property, as explained in Section 3. A typical application of enlarges/2 is given in the next clause.

Consider now the sum function from $\mathbb{R} \times \mathbb{R}$ to $\mathbb{R}$. The corresponding set map assigns to a pair (I_2, I_3) in $\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathbb{R})$ the set

$$I_2 + I_3 = \{r \in \mathbb{R} \mid r = x + y \text{ for some } x \in I_2\ , y \in I_3\} \tag{5.3}$$

in $\mathcal{P}(\mathbb{R})$. The following clause defines a bound on this map:

```
i(X1,Y1) contains i(X2,Y2)+i(X3,Y3):-
 X is X2+X3,
 Y is Y2+Y3,
 i(X1,Y1) enlarges i(X,Y),
 !.
```

As indicated in Section 4, the domain of this bound is the set of all pairs (I2, I3) in $\mathrm{std}(\mathbb{R}) \times \mathrm{std}(\mathbb{R})$ for which I1 contains I2+I3 is true. In this case, the domain is determined by the predicate enlarges/2 which is used in order to compensate for possible rounding errors introduced by is/2, and to make sure that the returned result I1 is again a standard set.

The same description applies to our bound on the product of (sets of) real numbers, if $+$ is replaced by $*$. The bound itself is defined as follows:

```
i(X1,Y1) contains i(X2,Y2)*i(X3,Y3):-
 A is X2*X3,
 B is X2*Y3,
 C is Y2*X3,
 D is Y2*Y3,
 sort_numbers([A,B,C,D],[X,_,_,Y]),
 i(X1,Y1) enlarges i(X,Y),
 !.
```

Here, we have used that the product of two standard sets I_2 and I_3 is an interval, and that the endpoints of this interval can be computed as the maximum and minimum of four products made up from the endpoints of the intervals I_2 and I_3. As in the case of the absolute value function, we find the maximum and the minimum by sorting.

We note that the difference function can be defined in terms of the sum and the unary minus. Since bounds can be composed (in the sense of Section 4), it suffices to give a general rule for this case. Such general rules are given in Subsection 13.1. An analogous remark applies to the quotient function.

This brings us to the function "inverse" which maps a real number r in $\mathcal{D} = \mathbb{R} \setminus \{0\}$ to $1/r$. Its lift to $\mathcal{P}(\mathcal{D})$ is defined by setting

$$I \text{ inverse} = \{r \in \mathbb{R} \mid r = 1/x \text{ for some } x \in I\}, \tag{5.4}$$

for every $I \in \mathcal{P}(\mathcal{D})$. A bound on this map is defined by the following clause:

```
i(X1,Y1) contains (i(X2,Y2) inverse):-
 P is X2*Y2,
 P>0,
 X is 1/Y2,
 Y is 1/X2,
 i(X1,Y1) enlarges i(X,Y),
 !.
```

The purpose of the condition `P>0` is to restrict the domain of this bound to standard sets that do not contain the number zero.

Finally, we consider the function that maps a real number r to $r^{\mathtt{N}}$, where `N` is some (sufficiently small) non–negative integer. By convention, $r^0 = 1$ for all r. The lift to $\mathcal{P}(\mathbb{R})$ of this function can be obtained by composing $I \mapsto [I, \mathtt{i(1,1)}]$ with the `N`–th iterate $\mathtt{f}^{\mathtt{N}}$ of the map $\mathtt{f} : [I, J] \mapsto [I, I * J]$, followed by the projection $[I, J] \mapsto J$. In order to get a bound on $\mathtt{f}^{\mathtt{N}}$, it suffices to give here a bound on the map `f`, such as the one defined by

```
[I,J1] contains f of [I,J] :-
 J1 contains I*J,
 !.
```

The rest is taken care of by a general clause for powers (iterates) of a map, given in Subsection 13.1. A bound on $I \mapsto I^{\mathtt{N}}$ can now be defined as follows:

```
i(Xn,Yn) contains i(X,Y)**N:-
 [_,i(Xn,Yn)] contains f**N of [i(X,Y),i(1,1)],
 !.
```

This completes our discussion of the basic operations on the standard sets for $\mathbb{R}$. The next step will be to establish bounds for operations on polynomials, which are the simplest functions in the Banach space $\mathcal{B}$. However, in order to motivate our choices, we first reformulate the proof of Theorem 1.2 as a fixed point problem, so that it will become clear what operations are needed.

6. The Contraction Mapping Principle

We prove the existence of a solution to (1.1) by applying the contraction mapping principle to an operator $\mathcal{R}$ which has the desired solution as a fixed point. The operator $\mathcal{R}$ will be defined on an open ball

$$U_\beta(G_0) = \{G \in \mathcal{B}_0 \mid \|G - G_0\| < \beta\} \tag{6.1}$$

of radius $\beta = 2^{-18}$, centered at some polynomial $G_0 \in \mathcal{B}_0$ to be chosen later on. In Prolog, β is specified by the clause

```
radius_of_ball_in_function_space(Beta):-
 negative_power_of_two(Beta,18).
```

Let now G be a fixed but arbitrary function in $U_\beta(G_0)$. Then $\mathcal{R}(G)$ is defined through the following sequence of steps:

i) Find a solution of the Schröder equation (1.5)

$$F(z) - \frac{1}{\lambda}F(G(z)) = 0, \tag{6.2}$$

where $\lambda = G'(0)$. This equation will be solved by an iterative procedure (see Section 10). The result is a one–parameter family of solutions $F = c\mathcal{F}$, and we will see that $\mathcal{F}$ lies in $\mathcal{B}_0$ and that c can be chosen such that $F(-1+\lambda^2) = 1$.

ii) Compose $\mathcal{F}$ with the function $z \mapsto -1 + \lambda^2 + \lambda^2 z$ to obtain F_1,

$$F_1(z) = \mathcal{F}(-1 + \lambda^2 + \lambda^2 z). \tag{6.3}$$

We will show that F_1 is an element of $\mathcal{B}$.

iii) Define F_2 to be the square of F_1,

$$F_2(z) = F_1(z)^2. \tag{6.4}$$

iv) Normalize F_2 according to (1.6) and (1.7), by setting

$$F_3(z) = F_2(z)/F_2(0) - 1. \tag{6.5}$$

This function F_3 may seem like a natural candidate for $\mathcal{R}(G)$, since $F_3 = G$ would imply that the functions $F = c\mathcal{F}$ and G have the properties described in Theorem 1.2. However, the map $G \mapsto F_3$ is not a contraction: it has "one unstable direction" which is roughly parallel to the identity function. We thus add the following step:

v) Let $\widetilde{\lambda} = {F_3}'(0)$ and define $\mathcal{R}(G)$ by the equation

$$\mathcal{R}(G)(z) = F_3(z) - 1.72(\lambda - \widetilde{\lambda})z. \tag{6.6}$$

The value 1.72 has been found empirically; see also the footnote in Section 12. In Prolog we define (a standard set containing) this number with the predicate `contraction_factor/1`,

```
contraction_factor(X):-
 X contains i(172,172)/i(100,100).
```

We note that if G is a fixed point of $\mathcal{R}$, then equation (6.6) implies that $\lambda = \widetilde{\lambda} - 1.72(\lambda - \widetilde{\lambda})$. As a consequence we have $\widetilde{\lambda} = \lambda$, and thus $G = F_3$. This in turn implies that the functions F and G have the properties described in Theorem 1.2.

The remaining part of this paper will be devoted to the proof of the following theorem, which implies Theorem 1.2.

Theorem 6.1. *There exist two positive real numbers $\rho < 1$ and $\varepsilon < (1-\rho)\beta$, and a polynomial $G_0 \in \mathcal{B}_0$ of degree nine (all three defined below), such that the following holds. The operator $\mathcal{R}$ is well defined and continuously differentiable as a map from $U_\beta(G_0)$ to $\mathcal{B}_0$, and at any given point $G \in U_\beta(G_0)$, the operator norm of its derivative $D\mathcal{R}(G)$ is bounded by ρ,*

$$\|D\mathcal{R}(G)\| \le \rho \quad (\approx 0.94)\ . \tag{6.7}$$

Furthermore, $\mathcal{R}(G_0)$ lies within distance ε of G_0,

$$\|\mathcal{R}(G_0) - G_0\| \le \varepsilon \quad (\approx 1.3 \cdot 10^{-7})\ . \tag{6.8}$$

By the contraction mapping principle, this theorem implies that $\mathcal{R}$ has a fixed point G^* in $U_\beta(G_0)$, and that

$$\|G^* - G_0\| \le \frac{\varepsilon}{1-\rho}. \tag{6.9}$$

7. Bounds Involving Polynomials

Let V be the vector space of polynomials over the real numbers. In this section we define bounds on maps from V, or from $V \times V$, to V or to $\mathbb{R}$. Following the procedure outline in Section 4, we start by defining the standard sets for V.

Definition 7.1. *We define* $\mathrm{std}(V)$ *to be the collection of all sets* `p([I0,I1,···])` *of the form*

$$\texttt{p([I0,I1,}\cdots\texttt{])} = \{p \in V \mid p(x) = p_0 + p_1 \cdot x + \cdots\ , \quad p_0 \in \texttt{I0},\ p_1 \in \texttt{I1},\ \cdots\}\ , \tag{7.1}$$

with `[I0,I1,···]` *any finite list of elements of* $\mathrm{std}(\mathbb{R})$. *For the set containing only the polynomial $p \equiv 0$ we also use the notation* `p([])`.

We note that our program will use these standard sets only to represent polynomials of degree 18 or less. Thus, the reader may think of V as being restricted accordingly.

Our first bound involving polynomials deals with the map “unary minus”. The negative of the zero polynomial is the zero polynomial,

```
p([]) contains -p([]):-
 !.
```

The other cases are covered by the clause

```
p([H1|T1]) contains -p([H2|T2]):-
 H1 contains -H2,
 p(T1) contains -p(T2),
 !.
```

Namely, let $p : x \mapsto c + x \cdot q(x)$ be an element of the standard set `p([H2|T2])`. Then c and q are elements of the standard sets `H2` and `p(T2)`, respectively. The bound is based on the identity $-p(x) = (-c) + x \cdot (-q(x))$, i.e., `H1` contains $-c$ and `p(T1)` contains the polynomial $-q$.

Similarly, our bound on the (lifted) multiplication of a polynomial with a real number is based on the identity $r \cdot p(x) = r \cdot (c + x \cdot q(x)) = (r \cdot c) + x \cdot (r \cdot q(x))$. The zero polynomial multiplied by a number is equal to the zero polynomial,

```
p([]) contains i(_,_)*p([]):-
 !.
```

The remaining cases are dealt with by the clause [*)]

```
p([H1|T1]) contains i(X,Y)*p([H2|T2]):-
 H1 contains i(X,Y)*H2,
 p(T1) contains i(X,Y)*p(T2),
 !.
```

The bound on the addition of two polynomials is again very similar. But there are two possible final stages in the recursive evaluation of `p(L2)+p(L3)`. If the list `L2` contains at least as many coefficients as the list `L3`, then we will end up with the zero polynomial in the second argument of the sum,

```
p(L) contains p(L)+p([]):-
 !.
```

Otherwise, the recursion will terminate with the zero polynomial in the first argument,

```
p(L) contains p([])+p(L):-
 !.
```

If none of the arguments matches `p([])`, then the following clause is used:

```
p([H1|T1]) contains p([H2|T2])+p([H3|T3]):-
 H1 contains H2+H3,
 p(T1) contains p(T2)+p(T3),
 !.
```

In order to bound the product of two polynomials, we will inductively reduce the first factor until the zero polynomial is reached and the clause

```
p([]) contains p([])*p(_):-
 !.
```

[*)] We use here the notation `i(X,Y)` and not simply `I`, or similar, in order to avoid that this argument matches terms for which it is not intended, such as `p(T)`, which would result in the clause being applied wrongly to the product of two polynomials. Our general strategy is to use the most elegant notation that does not lead to a conflict between different clauses of the same predicate.

applies. The reduction procedure is based on the fact that $(c+x\cdot q(x))\cdot h(x)=c\cdot h(x)+x\cdot(q(x)\cdot h(x))$. The corresponding clause is

```
p(L1) contains p([H2|T2])*p(L2):-
 p(L) contains p(T2)*p(L2),
 p(L1) contains H2*p(L2)+p([i(0,0)|L]),
 !.
```

We note that none of the clauses defined so far applies to a goal of the form `A contains B*C+Q`. This "problem" can be solved by reducing the given goal to `P contains B*C, A contains P+Q`. General clauses that carry out such reductions are given in Subsection 13.2.

Another operation which will be needed later assigns to a polynomial its `N`+1st Taylor coefficient. The zero polynomial has all coefficients equal to zero,

```
i(0,0) contains coeff(_) of p([]):-
 !.
```

In the other cases, a bound for the map $p\mapsto p(0)$ is given by

```
C contains coeff(0) of p([C|_]):-
 !.
```

and the higher coefficients are covered recursively by the clause

```
C contains coeff(N) of p([_|T]):-
 M is N-1,
 C contains coeff(M) of p(T),
 !.
```

Next, we consider the derivative of polynomials. For the zero polynomial we have

```
p([]) contains der(p([])):-
 !.
```

In order to handle the other cases in a reasonably efficient way, let us introduce the operator $\mathtt{b}=x\frac{d}{dx}$. By using the bound on $\mathtt{b}+\mathtt{0}\equiv\mathtt{b}$ given below, we can complete our bound on the derivative as follows:

```
p(T) contains der(p(L)):-
 p([_|T]) contains (b+0) of p(L),
 !.
```

Consider now the sum of `b` and `N` times the identity, in symbols `b+N`, which maps a monomial m_k of degree k to $(k+\mathtt{N})m_k$, for every $k\geq 0$. Here, `N` is any (sufficiently small) non–negative integer. Independently of `N`, the zero polynomial is mapped to itself by `b+N`,

```
p([]) contains (b+_) of p([]):-
 !.
```

In the remaining cases, the following clause applies:

```
p([H1|T1]) contains (b+N) of p([H2|T2]):-
 H1 contains i(N,N)*H2,
 M is N+1,
 p(T1) contains (b+M) of p(T2),
 !.
```

Namely, if $p(x) = c + x \cdot q(x)$ then $((\mathtt{b} + \mathtt{N})p)(x) = \mathtt{N}c + x \cdot ((\mathtt{b} + \mathtt{N} + 1)q)(x)$.

Finally, we will give a bound on the norm functional on V, which assigns to a polynomial p with coefficients $p_0, p_1, \ldots, p_n$ the number

$$\|p\| = \sum_{i=0}^{n} |p_i| \cdot D^i, \tag{7.2}$$

where $D = 1.7$ is the radius of the common domain of analyticity for functions in $\mathcal{B}$, as specified in Definition 1.1. In Prolog, we define a standard set containing D with the predicate `domain/1`,

```
domain(D):-
 D contains i(17,17)/i(10,10),
 !.
```

The norm of the zero polynomial is zero

```
i(0,0) contains norm(p([])):-
 !.
```

and the remaining cases are covered by the clause

```
i(0,Y) contains norm(p([H|T])):-
 domain(D),
 i(_,Y) contains abs(H)+D*norm(p(T)),
 !.
```

We note that this bound could be "improved" by substituting `i(X,Y)` for both `i(_,Y)` and `i(0,Y)` in the last clause. But for convenience later on, we ignore lower bounds on the norm of polynomials and replace them by the trivial lower bound $\mathtt{X} = \mathtt{0}$.

8. Approximate Computations with Polynomials

In this section we develop the tools for computing approximate solutions of the Schröder equation (6.2), and of a generalized version of this equation (see below) which includes an inhomogeneous term. These approximate solutions will be used later as input to a procedure that constructs the real solutions.

Consider now the inhomogeneous Schröder equation

$$F(z) - \frac{1}{\lambda} F(G(z)) = J(z), \tag{8.1}$$

where G and J are given functions, with $G(0) = J(0) = J'(0) = 0$, and where $\lambda = G'(0)$. If we define an operator $j(G, F)$ by setting

$$j(G, J)(F) = J + \frac{1}{\lambda} F \circ G, \tag{8.2}$$

then the equation (8.1) can be written in the form $F = j(G, J)(F)$. This suggests that a solution may be obtained by iteration, e.g. by taking the limit as $n \to \infty$ of the functions

$$f_n = j(G, J)^n (J + \kappa I) = \sum_{i=0}^{n} \frac{1}{\lambda^i} J \circ G^i + \frac{\kappa}{\lambda^n} G^n. \tag{8.3}$$

Here, G^i denotes the i–fold composition of G with itself, I is the identity function $I(z) = z$, and κ is an arbitrary constant (as the equation for F is affine). In Section 10 we will show that under suitable assumptions on G and J, the functions f_n converge in $\mathcal{B}_0$ to a solution of (8.1). Thus, f_n is a natural candidate for an approximate solution of the (inhomogeneous) Schröder equation, if n is large.

Our goal in this section is to compute f_n approximately, in the case where G and J are polynomials. To this end, we can freely combine bounds from previous sections with steps that only produce approximate results.

In particular, we may collapse any interval `i(X,Y)` to an interval of length zero, centered near the arithmetic mean of `X` and `Y`. To carry this out for every coefficient–interval in a set of polynomials, we will use the predicate `collapse/2` defined by

```
collapse(p([i(X,Y)|T1]),p([i(Z,Z)|T2])):-
 Z is (X+Y)/2,
 collapse(p(T1),p(T2)),
 !.
```

where the recursion ends with the zero polynomial

```
collapse(p([]),p([])):-
 !.
```

Another useful approximation splits a polynomial p into a polynomial q of degree $\leq$ `N` and a remainder r, according to the equation $p(x) = q(x) + x^{\mathtt{N+1}} \cdot r(x)$, and then discards r. The splitting is implemented with a predicate `split/4` whose first, third, and fourth argument is for the set containing p, q, and r, respectively. We simply take the list of coefficients from the first argument, say of length l_p , and split it into a "head" of length $l_q = \max(l_p, \mathtt{N}+1)$ and a tail of length $l_r = l_p - l_q$. This is done recursively by using the clause

```
split(p([H|T1]),N,p([H|T2]),p(L)):-
 N>=0,
 M is N-1,
 split(p(T1),M,p(T2),p(L)),
 !.
```

provided that l_q is nonzero. Otherwise the following clause applies, which is also reached at the end of the recursion:

```
split(p(L),_,p([]),p(L)):-
 !.
```

The two approximation steps will now be combined. For reasons that will become clear later, we choose the degree `N` for the above–mentioned truncation to be equal to 9. In Prolog,

```
degree(9).
```

The following clause specifies a way of collapsing a set of polynomials `p(L2)` into a set `p(L1)` that contains a single polynomial of degree ≤ 9:

```
p(L1) approximates p(L2):-
 degree(N),
```

```
 split(p(L2),N,p(L3),_),
 collapse(p(L3),p(L1)),
 !.
```

As a first step towards a numerical computation of f_n , we implement the composition of two polynomials p and r. Let $p(x) = c + x \cdot q(x)$. Then $(p \circ r)(x) = c + r(x) \cdot (q \circ r)(x)$, and this expression can be used to (compute as well as to) approximate $p \circ r$. Namely,

```
p(L1) approximates p([H|T]) o p(L2):-
 p(L3) approximates p(T) o p(L2),
 p(L4) contains p([H])+p(L3)*p(L2),
 p(L1) approximates p(L4),
 !.
```

where `H` contains c, `p(T)` contains q, and `p(L2)` contains r. The recursion ends with the composition of the zero polynomial with an arbitrary polynomial,

```
p([]) approximates p([]) o p(_):-
 !.
```

Next, we approximate the image of a polynomial F under map $j(G, J)$. By following the definition (8.2), we set

```
p(Fh) approximates j(p(G),p(J)) of p(F):-
 p(Fg) approximates p(F) o p(G),
 p(F1) contains p(J)+ p(Fg)/ (coeff(1) of p(G)),
 p(Fh) approximates p(F1),
 !.
```

Eventually, this approximate version of $j(G, J)$ needs to be iterated, according to (8.3). But this can be done by using a general clause given in Subsection 13.1.

In the case where $J \equiv 0$, we can increase precision and efficiency by choosing $n = 2^m$ and $\kappa = \lambda^n$, so that $f_n = q^m(G)$, where q is the operator defined by the equation

$$q(G) = G \circ G. \tag{8.4}$$

Again, it suffices to define an approximation for the operator q,

```
p(F) approximates q of p(G):-
 p(F) approximates p(G) o p(G),
 !.
```

9. Bounds Involving Analytic Functions

The bounds defined in this section involve functions in the space $\mathcal{B}$. We start by introducing the standard sets for this space.

Definition 9.1. std($\mathcal{B}$) *is defined to be the collection of all sets* `f(P,e(G,H))` *of the form*

$$\texttt{f(P,e(G,H))} = \{F \in \mathcal{B} | F(z) = p(z) + z \cdot g(z) + z^{10} \cdot h(z),\ p \in \texttt{P}, \|g\| \in \texttt{G}, \|h\| \in \texttt{H}\}, \quad (9.1)$$

with $\texttt{P} \in \mathrm{std}(V)$*, and* `G`*,* `H` *intervals in* $\mathrm{std}(\mathbb{R})$ *with lower boundaries equal to zero (as we have no need for nontrivial lower bounds on norms).*

In what follows, if a function $F \in \mathcal{B}$ is specified as in (9.1), then g will be referred to as the "general error" for F, and h as the "higher order error" for F. We note that p will always be of degree less than 19, and usually of degree less than 10.

Given that any function in $\mathcal{B}$ has a unique representation as the sum of a polynomial of degree less than ten, and a function $\varphi \in \mathcal{B}$ with $\varphi(z) = \mathcal{O}(z^{10})$, the reader may wonder why we consider representations with nonzero general error terms. The reason for our choice has to do with the following fact. Let ψ be some arbitrary analytic function that maps the disk $|z| < D$ to itself, and consider the image under $\varphi \mapsto \varphi \circ \psi$ of a standard set `f(p([]),e(i(0,0),i(0,R)))` with $\texttt{R} > 0$. It is easy to show that this image is contained in the ball $B = \{f \in \mathcal{B} :\ \|f\| \le \texttt{R}\}$, but not much more can be said in general. Thus, in order to bound the set map associated with $\varphi \mapsto \varphi \circ \psi$, we need to find a standard set $S \subset \mathcal{B}$ that contains B. This can be done reasonably well with S of the form `f(p([C]),e(G,i(0,0)))` (our choice of multiplying general errors with z is a disadvantage here, but it turns out to be convenient later). By contrast, bounds that use only standard sets S of the form `f(P,e(i(0,0),H))` are substantially weaker. That is, if S contains B then S also contains functions whose norm is much larger than `R`. Another solution to this problem, besides introducing general error terms, is to use a Banach space (of analytic functions) with a weighted sup–norm on the sequence of Taylor coefficients; see e.g. [KW7].

Our first bound in this section is on (the set map associated with) the norm functional on $\mathcal{B}$. The domain of this bound is a subset of std($\mathcal{B}$), and its range is a set of intervals in std($\mathbb{R}$) with lower boundaries equal to zero. By using the inequality

$$0 \le \|f\| \le \|p\| + D \cdot \|g\| + D^{10} \cdot \|h\|, \quad (9.2)$$

such a bound can be defined as follows:

```
i(0,Y) contains norm(f(P,e(G,H))):-
 domain(D),
 degree(N),
 N1 is N+1,
 i(_,Y) contains norm(P)+D*G+D**N1*H,
 !.
```

We will also need bounds for the two functionals $F \mapsto F(0)$ and $F \mapsto F'(0)$. If $F \in \mathcal{B}$ is represented as in (9.1), then $F(0) = p(0)$ and $F'(0) = p'(0) + g(0)$. Thus,

```
S contains coeff(0) of f(P,_):-
```

```
 S contains coeff(0) of P,
 !.
```

and, since $|g(0)| \leq \|g\|$, we have

```
S contains coeff(1) of f(P,e(i(_,Y),_)):-
 X is -Y,
 S contains (coeff(1) of P)+i(X,Y),
 !.
```

The following three bounds are based on elementary properties of the norm. The property $\|c \cdot f\| = |c| \cdot \|f\|$ applies to the multiplication of a function by a number

```
f(P1,e(G1,H1)) contains i(X,Y)*f(P2,e(G2,H2)):-
 P1 contains i(X,Y)*P2,
 G1 contains abs(i(X,Y))*G2,
 H1 contains abs(i(X,Y))*H2,
 !.
```

as well as to the unary minus

```
f(P,E) contains -f(P1,E):-
 P contains -P1,
 !.
```

and the triangle inequality is used when adding two functions,

```
f(P1,e(G1,H1)) contains f(P2,e(G2,H2))+f(P3,e(G3,H3)):-
 P1 contains P2+P3,
 G1 contains G2+G3,
 H1 contains H2+H3,
 !.
```

The multiplication of two functions is slightly more complicated. Consider functions f_2 and f_3 of the form

$$\begin{aligned} f_2(z) &= p_2(z) + z \cdot g_2(z) + z^{10} \cdot h_2(z), \\ f_3(z) &= p_3(z) + z \cdot g_3(z) + z^{10} \cdot h_3(z), \end{aligned} \tag{9.3}$$

with p_2 and p_3 polynomials. Define a polynomial p_1 of degree less than ten, and a polynomial r, by the equation

$$p_1(x) + x^{10} \cdot r(x) = p_2(x) \cdot p_3(x). \tag{9.4}$$

Then the product of f_1 and f_2 is given by the equation

$$\begin{aligned} f_2(z) \cdot f_3(z) = p_1(z) & \\ &+ z \cdot \big[p_2(z) \cdot g_3(z) + g_2(z) \cdot p_3(z) + z \cdot g_2(z) \cdot g_3(z)\big] \\ &+ z^{10} \cdot \big[r(z) + z^{10} \cdot h_2(z) \cdot h_3(z) + p_2(z) \cdot h_3(z) + h_2(z) \cdot p_3(z) \\ &\qquad + z \cdot g_2(z) \cdot h_3(z) + z \cdot h_2(z) \cdot g_3(z)\big]. \end{aligned} \tag{9.5}$$

This suggests the following bound. We start with the multiplication of the two sets of polynomials P2 $\ni p_2$ and P3 $\ni p_3$,

```
f(P1,e(i(0,Yg),i(0,Yh))) contains f(P2,e(G2,H2))*f(P3,e(G3,H3)):-
 P contains P2*P3,
```

which we split into P1 $\ni p_1$ and R $\ni r$ according to equation (9.4),

```
 degree(N),
 split(P,N,P1,R),
```

Then, by using that $\|f \cdot g\| \leq \|f\| \cdot \|g\|$ for any two functions f and g in $\mathcal{B}$ (this can be verified e.g. by using the Taylor expansion of $f \cdot g$), we compute an upper bound on the norm of the general error in (9.5),

```
 domain(D),
 i(_,Yg) contains norm(P2)*G3+ G2*norm(P3)+ D* (G2*G3),
```

and an upper bound on the norm of the higher order error in (9.5),

```
 N1 is N+1,
 i(_,Yh) contains norm(R)+ (D**N1)*H2*H3+ norm(P2)*H3+ H2*norm(P3)
                        + D*G2*H3+ D*H2*G3,
 !.
```

Consider now the composition $p \circ f$ of a function $f \in \mathcal{B}$ with a polynomial p. This operation can be carried out recursively: if $p(x) = c + x \cdot q(x)$, then we have $(p \circ f)(x) = c + f(x) \cdot (q \circ f)(x)$. In particular, if p is identically zero, then so is the result,

```
f(p([]),E) contains p([]) o f(_,_):-
 zero_error(E),
 !.
```

The predicate zero_error/1 used here has been introduced for convenience. Its only clause is

```
zero_error(e(i(0,0),i(0,0))).
```

The case of a general polynomial is covered by the clause

```
f(P1,E1) contains p([H|T]) o f(P2,E2):-
 zero_error(E),
 f(P1,E1) contains f(p([H]),E)+f(P2,E2)* (p(T) o f(P2,E2)),
 !.
```

where f(p([H]),E) contains the constant function $f(z) = c$.

Next, we discuss the composition of a function $f_3 \in \mathcal{B}$ with another function $f_2 \in \mathcal{B}$. In order to ensure that $f_2 \circ f_3$ is well defined, we require that $\|f_3\| < D$. This condition implies that f_3 maps the domain $|z| < D$ into itself, since

$$\sup_{|z|<D} |f(z)| \leq \|f\|, \tag{9.6}$$

for every $f \in \mathcal{B}$. Let now f_2 and f_3 be represented in the form (9.3). Then we have

$$(f_2 \circ f_3)(z) = (p_2 \circ f_3)(z) + c + z \cdot g(z), \tag{9.7}$$

where

$$c + z \cdot g(z) = r(z) = f_3(z) \cdot (g_2 \circ f_3)(z) + (f_3(z))^{10} \cdot (h_2 \circ f_3)(z). \tag{9.8}$$

The first term on the right hand side of (9.7) is of the form discussed earlier. For the remainder r we have

$$\|r\| \leq \|f_3\| \cdot \|g_2\| + \|f_3\|^{10} \cdot \|h_2\|. \tag{9.9}$$

This bound can be used to estimate the constant c, and the function g which contributes to the general error for $f_2 \circ f_3$. Namely, from the definition of the norm in $\mathcal{B}$, we see that

$$-\|r\| \leq c \leq \|r\|, \tag{9.10}$$

and

$$0 \leq \|g\| \leq \|r\|/D. \tag{9.11}$$

We note that this is a place where the advantage of using general errors becomes apparent. Namely, if we had to convert (9.9) into eleven bounds (as opposed to two), by insisting on a representation $r(x) = c_0 + c_1 x + \ldots + c_9 x^9 + x^{10} h(x)$, then the result would be much (as opposed to slightly) weaker than the original bound on $\|r\|$. In Prolog, the bound on composition is now defined as follows. By using the predicate `lt/2`,

```
i(_,Y) lt i(X,_):-
 Y<X,
 !.
```

which makes `I1 lt I2` true if every $y \in$ `I1` is smaller than every $x \in$ `I2`, we first check that the norm of the argument corresponding to the function f_3 is less than D,

```
f(P1,E1) contains f(P2,e(G2,H2)) o f(P3,E3):-
 domain(D),
 U contains norm(f(P3,E3)),
 U lt D,
```

Then, we use the estimate (9.9) to bound the norm of the remainder r,

```
 degree(N),
 N1 is N+1,
 i(_,Y) contains U*G2+U**N1*H2,
```

and convert the result according to (9.10) and (9.11) into a bound `C` and a bound `G` (both represented as a standard set for $\mathcal{B}$),

```
 X is -Y,
 zero_error(E),
 C=f(p([i(X,Y)]),E),
 Eg contains i(0,Y)/D,
 G=f(p([]),e(Eg,i(0,0))),
```

At the end, the pieces are added up as in (9.7),

```
 f(P1,E1) contains P2 o f(P3,E3) + C + G,
 !.
```

Finally, we consider the derivative of a function $f_2 \in \mathcal{B}$. Since f_2' is not necessarily an element of $\mathcal{B}$, we will compose f_2' with another function $f_3 \in \mathcal{B}$ whose norm is μD, with $\mu < 1$. Assume now again that f_2 and f_3 are given in the form (9.3). Then

$$(f_2' \circ f_3)(z) = (p_2' \circ f_3)(z) + c + z \cdot g(z), \tag{9.12}$$

where

$$c + z \cdot g(z) = r(z) = (e_2' \circ f_3)(z), \tag{9.13}$$

with

$$e_2(z) = z \cdot g_2(z) + z^{10} \cdot h_2(z). \tag{9.14}$$

In order to estimate the two terms that contribute to $e_2' \circ f_3$, let f be a function in $\mathcal{B}$ with $f(z) = \mathcal{O}(z^j)$, where $j > 0$. Then we have the bound

$$\|f' \circ f_3\| \le \|f'(\mu\,.)\| \le \sup_{i \ge j}(i\,\mu^{i-1})\,\|f\|/D. \tag{9.15}$$

The second inequality in (9.15) has been obtained by using the fact that the coefficient of z^{i-1} in the Taylor series for $f'(\mu z)$ is $i\mu^{i-1}$ times the i–th Taylor coefficient of f. As a result, the norm of the function r in equation (9.13) can be bounded as follows:

$$\|r\| \le \sup_{i \ge 1}(i\,\mu^{i-1}) \cdot \|g_2\| + \sup_{i \ge 10}(i\,\mu^{i-1}) \cdot D^9 \cdot \|h_2\|. \tag{9.16}$$

An upper bound on the supremum in (9.15) is given by the inequality

$$\sup_{i \ge j}(i\,\mu^{i-1}) \le \sum_{i \ge j} i\ \mu^{i-1} = \Big(\frac{j\,\mu^{j-1}}{1-\mu} + \frac{\mu^j}{(1-\mu)^2}\Big), \tag{9.17}$$

which is far from optimal, but easy to implement:

```
i(0,X) contains sup(J,Q):-
 J1 is J-1,
 Q1 contains i(1,1)-Q,
 i(_,X) contains (Q**J1)*i(J,J)/Q1 + (Q**J)/ (Q1**2),
 !.
```

We are now ready to define our bound on the set map for $(f_2, f_3) \mapsto f_2' \circ f_3$. The first step is to verify that $\mu < 1$,

```
f(P1,E1) contains der(f(P2,e(G2,H2))) o f(P3,E3):-
 domain(D),
 Mu contains norm(f(P3,E3))/D,
 Mu lt i(1,1),
```

Then, following (9.16), we compute a bound the norm of the error term r

```
 degree(N),
 N1 is N+1,
 i(_,Y) contains sup(1,Mu)*G2+ sup(N1,Mu)* (D**N)*H2,
```

and convert the result into a bound C and a bound G, in exactly the same way as we did in the clause for composition, by using (9.10) and (9.11),

```
X is -Y,
zero_error(E),
C=f(p([i(X,Y)]),E),
Eg contains i(0,Y)/D,
G=f(p([]),e(Eg,i(0,0))),
```

The last step is to add up the pieces according to equation (9.12),

```
P contains der(P2),
f(P1,E1) contains P o f(P3,E3) + C + G,
!.
```

10. The Schröder Equation

In this section we show how to obtain accurate bounds on solutions of the Schröder equation

$$F(z) - \frac{1}{\lambda} F(G(z)) = J(z). \tag{10.1}$$

The general procedure is independent of whether the equation is homogeneous ($J \equiv 0$) or not. Namely, we first compute a polynomial F_0 that "solves" the given equation to the desired degree of accuracy, by using the approximate methods from Section 8. By substituting the ansatz $F = F_0 + F_1$ into (10.1), we then obtain a Schröder equation for F_1, with a (small) inhomogeneous term

$$J_1 = J - \bigl(F_0 - \frac{1}{\lambda} F_0 \circ G\bigr). \tag{10.2}$$

Here, both J_1 and F_1 will be regarded as error terms, i.e., besides proving that the equation for F_1 has a solution, we restrict ourselves to estimating the norms of the general and higher order error for F_1 in terms of the corresponding norms for J_1. We start by explaining the estimates used in this last step.

Let G and J be given functions in $\mathcal{B}_0$ such that $\lambda = G'(0)$ is nonzero and $J(z) = \mathcal{O}(z^k)$, for some integer $k > 1$. Consider the corresponding sequence of functions f_n, defined by equation (8.3) for $\kappa = 0$. Below, we will show that this sequence converges in $\mathcal{B}_0$, provided that

$$(\|G\|/D)^2 / \, |\lambda| < 1. \tag{10.3}$$

We note that this condition implies that $\|G\| < D$ and $|\lambda| < 1$, as can be seen by using the bound $|\lambda| = |G'(0)| \le \|G\|/D$. This shows e.g. that the map $j(G, J)$, defined by (8.2), is continuous from $\mathcal{B}_0$ to $\mathcal{B}_0$. Thus, if we prove that the sequence $\{f_n\}$ converges in $\mathcal{B}_0$, then we know that the limit F satisfies the Schröder equation (10.1).

Assume now that the condition (10.3) is satisfied, and define two functions H and K in $\mathcal{B}$, by setting $H(z) = G(z)/z$ and $K(z) = J(z)/z^k$. In addition, let $H_0(z) = 1$ and

$$H_i(z) = \prod_{j=0}^{i-1} H(G^j(z)), \tag{10.4}$$

for all $i > 0$. Then, by using that $G^i(z) = z \cdot H_i(z)$, we can rewrite equation (8.3), with $\kappa = 0$, in the form

$$f_n(z) = z^k \cdot \sum_{i=0}^{n} \frac{1}{\lambda^i} [H_i(z)]^k (K \circ G^i)(z). \tag{10.5}$$

The estimate

$$\|H_i\| \le \|H\|^i = (\|G\|/D)^i, \tag{10.6}$$

together with (10.3), now implies that the sum (10.5) is bounded term by term by a convergent geometric series. Thus, the limit $F = \lim_{n\to\infty} f_n$ exists in $\mathcal{B}_0$, and

$$\|F\| \le D^k \sum_{i=0}^{\infty} \bigl((\|G\|/D)^k/|\lambda|\bigr)^i \|K\| \le \frac{\|J\|}{1-(\|G\|/D)^k/|\lambda|}. \tag{10.7}$$

Furthermore, since the Taylor coefficients of f_n converge to those of F, we have $F(z) = \mathcal{O}(z^k)$. In what follows, we will also use the symbol $y(G)J$ in place of F, i.e.,

$$y(G)J = F = \sum_{i=0}^{\infty} \frac{1}{\lambda^i} J \circ G^i. \tag{10.8}$$

The following Prolog clause implements the norm estimate (10.7); it makes a goal `E contains norms(y(G) of J)` true only if the Schröder equation with $G \in$ `G` and $J \in$ `J` admits a solution $F \in$ `f(p([]),E)` of the form described above. Here, we assume that $G \in \mathcal{B}_0$ and that $J \in \mathcal{B}_1 = \{f \in \mathcal{B} : f(0) = f'(0) = 0\}$, i.e., we regard `G` and `J` as standard sets for the spaces $\mathcal{B}_0$ and $\mathcal{B}_1$, respectively. After checking that G satisfies the condition (10.3),

```
e(Fg,Fh) contains norms(y(G) of f(Jp,e(Jg,Jh))):-
 L contains abs(coeff(1) of G),
 domain(D),
 X contains norm(G)/D,
 C contains X*X/L,
 C lt i(1,1),
```

we split the function J into a term $z \mapsto J_p(z) + z \cdot J_g(z)$ of order z^2, and a higher order error $z \mapsto z^{10} \cdot J_h(z)$. The images under $y(G)$ of the two parts are estimated by using (10.7) first with $k = 2$,

```
 Fg contains (Jg+norm(Jp)/D)/ (i(1,1)-C),
```

and then with $k = 10$,

```
 degree(N),
 N1 is N+1,
 Fh contains Jh/ (i(1,1)-X**N1/L),
 !.
```

We are now prepared to bound the set map for $(G, J) \mapsto y(G)J$, in the way described at the beginning of this section. Again, this bound is restricted to (standard sets representing) functions G and J that satisfy $G(0)=0$ and $J(0)=J'(0)=0$. The first step is to compute an approximate polynomial solution F_0,

```
f(P1,E1) contains y(f(P2,E2)) of f(Jp,e(Jg,Jh)):-
 Pj approximates Jp,
 P1 approximates j(P2,Pj)**16 of Pj,
 zero_error(E),
 F0=f(P1,E),
```

Then, a standard set is determined that contains the function J_1 defined in (10.2),

```
 L contains coeff(1) of f(P2,E2),
 J1 contains f(Jp,e(Jg,Jh))- (F0- (F0 o f(P2,E2))/L),
```

We note that $J_1(z)=\mathcal{O}(z^2)$, since $F_0(0)=0$ by construction, and $\lambda=G'(0)$. Thus, the estimate (10.7) can be applied to bound the norms of the error terms for F_1,

```
 E1 contains norms(y(f(P2,E2)) of J1),
 !.
```

Consider now the homogeneous Schröder equation, with G and λ as above. Given a function $F_0\in\mathcal{B}_0$ such that $F_0'(0)=1$, define J_1 by (10.2), and denote by $\eta(G)$ the solution of (10.1) obtained by solving the equation $F_1-\frac{1}{\lambda}F_1\circ G=J_1$ for $F_1=F-F_0$ as described earlier, i.e.,

$$\eta(G)=F=F_0+y(G)\bigl(-F_0+\frac{1}{\lambda}F_0\circ G\bigr). \qquad (F_0'(0)=1) \qquad (10.9)$$

We note that this defines $\eta(G)$ independently of the choice of F_0. Namely, if ΔF is the difference between two solutions (of the homogeneous Schröder equation) whose derivatives agree at the origin, then ΔF is a solution that satisfies $\Delta F(z)=\mathcal{O}(z^2)$, which, as is easy to see, implies that $\Delta F\equiv 0$.

A bound on the set map associated with η can be defined as follows: first, we compute an approximate polynomial solution F_0 that satisfies $F_0'(0)=1$,

```
f(P1,E1) contains eta of f(P2,E2):-
 P approximates q**6 of P2,
 C contains coeff(1) of P,
 P1 contains P/C,
 zero_error(E),
 F0=f(P1,E),
```

and then we continue as in the previous clause,

```
 L contains coeff(1) of f(P2,E2),
 J1 contains - (F0- (F0 o f(P2,E2))/L),
 E1 contains norms(y(f(P2,E2)) of J1),
 !.
```

11. Bounds on Maps and Operators

Our main goal in this section is to give a bound on the operator $\mathcal{R}$. Following the (informal) five–step definition given in Section 6, we will write $\mathcal{R}$ as the composition of five maps, $\mathcal{R} = k \circ n \circ s \circ c \circ h$. Since several of these maps involve the number $\lambda = G'(0)$, we have chosen to use the space $\mathcal{B} \times \mathbb{R}$ for intermediate results. The direct product of a standard set `F` for $\mathcal{B}$ with a standard set `I` for $\mathbb{R}$ will be denoted by `x(F,I)`.

We start with the map h (solving the <u>h</u>omogeneous Schröder equation) from $\mathcal{B}_0$ to $\mathcal{B} \times \mathbb{R}$, which is defined by the equation

$$h(G) = \begin{pmatrix} F \\ \lambda \end{pmatrix}, \tag{11.1}$$

where $\lambda = G'(0)$, and where F is the solution of the equation $F - \frac{1}{\lambda} F \circ G = 0$, as described in the previous section. The corresponding bound is given by

```
x(F,L) contains h of G:-
 F contains eta of G,
 L contains coeff(1) of G,
 !.
```

In this clause, `G` and `F` are considered standard sets for $\mathcal{B}_0$, defined as in (9.1), but with $\mathcal{B}$ replaced by $\mathcal{B}_0$.

Next, we define the operator c (<u>c</u>omposition) on $\mathcal{B} \times \mathbb{R}$ by setting

$$c \begin{pmatrix} F \\ \lambda \end{pmatrix}(z) = \begin{pmatrix} F(-1 + \lambda^2 + \lambda^2 z) \\ \lambda \end{pmatrix}. \tag{11.2}$$

Here, and in what follows, the second component λ is being identified with the constant map $z \mapsto \lambda$, in order to simplify notation. A bound on c is defined by the clause

```
x(F1,L) contains c of x(F,L):-
 B contains L*L,
 A contains B-i(1,1),
 zero_error(E),
 F1 contains F o f(p([A,B]),E),
 !.
```

We also need the operator s (<u>s</u>quaring a function),

$$s \begin{pmatrix} F_1 \\ \lambda \end{pmatrix}(z) = \begin{pmatrix} F_1(z)^2 \\ \lambda \end{pmatrix}, \tag{11.3}$$

and the corresponding bound

```
x(F2,L) contains s of x(F1,L):-
 F2 contains F1*F1,
 !.
```

as well as the operator n (<u>n</u>ormalization),

$$n\begin{pmatrix} F_2 \\ \lambda \end{pmatrix}(z) = \begin{pmatrix} F_2(z)/F_2(0) - 1 \\ \lambda \end{pmatrix}, \tag{11.4}$$

and its bound

```
x(f(p([i(0,0)|P3]),E3),L) contains n of x(F2,L):-
 S contains coeff(0) of F2,
 f(p([_|P3]),E3) contains F2/S,
 !.
```

The last step in this sequence is the map k (<u>k</u>ey to contraction) from $\mathcal{B}_0 \times \mathrm{I\!R}$, which is the range of n, to the space $\mathcal{B}_0$,

$$k\begin{pmatrix} F_3 \\ \lambda \end{pmatrix}(z) = F_3(z) + 1.72 \cdot (\lambda - \widetilde{\lambda}) \cdot z, \tag{11.5}$$

where $\widetilde{\lambda} = F_3'(0)$. A bound on this map k is given by the clause

```
f(P,E3) contains k of x(f(P3,E3),L):-
 L3 contains coeff(1) of f(P3,E3),
 contraction_factor(S),
 B contains (L-L3)*S,
 P contains P3+p([i(0,0),B]),
 !.
```

This leads to the following bound on the operator $\mathcal{R}$:

```
G1 contains r of G:-
 G1 contains (k o n o s o c o h) of G,
 !.
```

where the composition of two bounds is defined by

```
Z contains (F o G) of X:-
 Y contains G of X,
 Z contains F of Y,
 !.
```

Finally, to give a proper definition of the operator $\mathcal{R}$, we define its domain $\mathcal{D}_{\mathcal{R}}$ by requiring that the condition (10.3) be satisfied, that the function $z \mapsto -1 + \lambda^2 + \lambda^2 z$ be of norm less than D, and that $F_2(0) \neq 0$.

12. Derivatives

The derivative $D\mathcal{R}$, if it exists, of the operator $\mathcal{R}$, maps a pair (G, H) from $\mathcal{D}_{\mathcal{R}} \times \mathcal{B}_0$ to the function $H_1 = D\mathcal{R}_G H$ in $\mathcal{B}_0$, where $D\mathcal{R}_G$ denotes the (Fréchet) derivative of $\mathcal{R}$ at G. By the definition of $\mathcal{R}$, we have

```
H1 contains (d(r) at G) of H:-
 H1 contains (d(k o n o s o c o h) at G) of H,
 !.
```

In this section we will show that the maps k, n, s, c, and h are of class C^1 on their domains, and we will define bounds for their derivatives (and thus for their tangent maps, using the bounds from the last section). Consequently, $\mathcal{R}$ is of class C^1 on its domain, and a bound for $D\mathcal{R}$ is obtained by using the chain rule

```
D contains (d(F o G) at X) of H:-
 Y contains G of X,
 D contains ((d(F) at Y) o (d(G) at X)) of H,
 !.
```

Here, and in what follows, we assume that the space $\mathcal{B} \times \mathbb{R}$ has been equipped with a norm that makes the projection onto the first coordinate an isometry.

We start with the map h that solves the Schröder equation. Its domain $\mathcal{D}_h \subset \mathcal{B}_0$ is defined by the condition (10.3). On this domain, the map $G \mapsto F \circ G$ is of class C^1 (and even analytic) for every function $F \in \mathcal{B}$. This fact, together with the implicit function theorem, implies that the map h is C^1 on $\mathcal{D}_h$. The derivative of h at G can be written as

$$Dh_G K = \begin{pmatrix} H \\ \mu \end{pmatrix}, \tag{12.1}$$

with $\mu = K'(0)$, and $H = y(G)J$ the solution of the equation $H - (H \circ G)/\lambda = J$, where $J = ((F' \circ G) \cdot K - \mu \cdot F)/\lambda$, and where F and λ are as in (11.1). Our bound for Dh is defined as follows:

```
x(H,Mu) contains (d(h) at G) of K:-
 Mu contains coeff(1) of K,
 x(F,L) contains h of G,
 Fp contains der(F) o G,
 J contains (Fp*K-Mu*F)/L,
 H contains y(G) of J,
 !.
```

Next, we consider the operator c. It is straightforward to check that this operator is C^1 on the domain defined by the condition $\|\psi\| < D$, where $\psi(z) = -1 + \lambda^2 + \lambda^2 z$, and that its derivative is given by

$$Dc_{\begin{pmatrix} F \\ \lambda \end{pmatrix}} \begin{pmatrix} H \\ \mu \end{pmatrix}(z) = \begin{pmatrix} H(\psi(z)) + F'(\psi(z)) \cdot (2\lambda + 2\lambda z) \cdot \mu \\ \mu \end{pmatrix}. \tag{12.2}$$

The following defines a bound on Dc:

```
x(H1,Mu) contains (d(c) at x(F,L)) of x(H,Mu):-
 B contains L*L,
 A contains B-i(1,1),
 zero_error(E),
 Psi=f(p([A,B]),E),
 Fp contains der(F) o Psi,
 C contains i(2,2)*L*Mu,
 Fh=f(p([C,C]),E),
 H1 contains H o Psi + Fp*Fh,
 !.
```

Squaring a function is clearly a C^1 operator on $\mathcal{B}\times\mathbb{R}$. For the derivative of s we have the expression

$$Ds_{\begin{pmatrix}F_1\\ \lambda\end{pmatrix}}\begin{pmatrix}H_1\\ \mu\end{pmatrix}(z)=\begin{pmatrix}2\,F_1(z)\,H_1(z)\\ \mu\end{pmatrix}, \tag{12.3}$$

and the corresponding bound is

```
x(H2,Mu) contains (d(s) at x(F1,_)) of x(H1,Mu):-
 H2 contains i(2,2)*F1*H1,
 !.
```

The normalization operator n and its derivative are (by the chain rule, which was also used above) well defined and continuous on the domain $F_2(0)\neq 0$. A bound on Dn is obtained by using the formula

$$Dn_{\begin{pmatrix}F_2\\ \lambda\end{pmatrix}}\begin{pmatrix}H_2\\ \mu\end{pmatrix}(z)=\begin{pmatrix}H_2(z)/F_2(0)-H_2(0)F_2(z)/F_2(0)^2\\ \mu\end{pmatrix}, \tag{12.4}$$

which translates to

```
x(H3,Mu) contains (d(n) at x(F2,_)) of x(H2,Mu):-
 S contains coeff(0) of F2,
 T contains coeff(0) of H2,
 H3 contains H2/S - T*F2/ (S*S),
 !.
```

As for the operator k, it suffices to say that it is linear and continuous. Thus,

```
K contains (d(k) at x(_,_)) of x(H3,Mu):-
 K contains k of x(H3,Mu),
 !.
```

independent of the base point. This concludes the definition of our bound for the derivative of $\mathcal{R}$.

What we need now is a bound for the operator norm of $D\mathcal{R}$. We start with a rough estimate for the operator norm of $L\pi_{10}$, where L is some continuous linear operator on $\mathcal{B}_0$, and where $\pi_0,\pi_1,\ldots$ are the projections on $\mathcal{B}$ defined by

$$(\pi_j f)(z)=\sum_{n=j+1}^{\infty}\frac{f^{(n)}(0)}{n!}z^n. \tag{12.5}$$

Denoting by B_j the unit ball in the subspace $\pi_j\mathcal{B}$, we have

$$\|L\pi_j\| = \sup_{h\in B_j} \|Lh\|. \tag{12.6}$$

Given a bound for L, the following clause defines an interval `i(0,Yj)` in std($\mathbb{R}$) that contains the norm (12.6) in the case $j = 10$:

```
i(0,Yj) contains operator_norm(L o pi(J)):-
 degree(N),
 J is N+1,
 domain(D),
 Di contains D inverse,
 i(_,Y) contains Di**J,
 H=f(p([]),e(i(0,0),i(0,Y))),
 i(0,Yj) contains norm(L of H),
 write_useful_information(J,Yj),
 !.
```

In order to give the corresponding estimate in the cases $j < 10$, we will use the fact that

$$\|L\pi_j\| = \max\{\|Lh_j\|, \|L\pi_{j+1}\|\}, \tag{12.7}$$

where $h_j(z) = (z/D)^j$. This identity holds since the map $u \mapsto \sum_{j\geq 1} u_j h_j$ is an isometry from ℓ^1 onto $\mathcal{B}_0$. The functions (basis vectors) h_j can be constructed by applying the appropriate power of the operator ζ,

$$(\zeta f)(z) = (z/D)f(z), \tag{12.8}$$

to the function $h_0 \equiv 1$. A bound for this operator ζ is given by the clause

```
H1 contains z of H0:-
 domain(D),
 Di contains D inverse,
 zero_error(E),
 H1 contains f(p([i(0,0),Di]),E)*H0,
 !.
```

Thus, by using the identity (12.7), we have

```
i(0,Y) contains operator_norm(L o pi(J)):-
 zero_error(E),
 H0=f(p([i(1,1)]),E),
 Hj contains (z**J) of H0,
 i(0,Yj) contains norm(L of Hj),
 write_useful_information(J,Yj),
 J1 is J+1,
 i(0,Z) contains operator_norm(L o pi(J1)),
 max(Y,Yj,Z),
```

```
    !.
```

where Y is the maximum of the numbers Yj and Z, according to the clause

```
max(X,X,Y):-
  X>=Y,
  !.
```

and

```
max(Y,X,Y):-
  Y>X,
  !.
```

We have also used the predicate `write_useful_information/2`, which writes information about the norm of the J–th basis vector to the output (the quoted strings used here are atoms),

```
write_useful_information(J,Yj):-
  write('norm of L(h'),write(J),write(') is approx.= '),
  write(Yj),nl.
```

Finally, the following clause defines a bound `G` $\mapsto$ `X` for the map $G \mapsto \|D\mathcal{R}_G\|$, if used with `L=(d(r) at G)`:

```
X contains operator_norm(L):-
  X contains operator_norm(L o pi(1)),
  !.
```

At this point most of the bounds which we need to prove Theorem 6.1 have been constructed.

We note that the operator norm of $D\mathcal{R}_G$ is determined by a single "column" vector $D\mathcal{R}_G h_j$. Thus, in order to get an accurate bound on $\|D\mathcal{R}_G\|$, the crucial column should be among the ones that are estimated explicitly, and the corresponding estimate should be sufficiently accurate. These two requirements determined to a large extent our choice of `degree(N)` [*)]. Here, it is of course useful that (the operator $F \mapsto F \circ \psi$, and thus) the operator $D\mathcal{R}_G$ is compact, for every G in the domain $\mathcal{D}_{\mathcal{R}}$.

13. General Clauses

In this section we have collected Prolog clauses which are necessary for the functioning of the program, but which are of very general nature and to some extent independent of our problem. The reader (but not Prolog) can safely skip this section at first reading.

[*)] The value `N=9` has been found by trial and error. Another quantity which has been chosen empirically, is the domain $|z| < D$ that enters the definition of $\mathcal{B}$. Typically, increasing D makes the first few of the vectors $D\mathcal{R}_G h_i$ larger in norm, and the others smaller. Similarly, varying the "contraction factor" affects the norm of these vectors, although mainly for small i. Our choice of these parameters has been such as to optimize our estimate on the norm of $D\mathcal{R}_G$.

13.1. Differences, Quotients, and Powers

The following clause defines a bound for the difference of two standard sets in terms of the corresponding bounds on the unary minus and the sum:

```
X contains Y-Z:-
 X contains Y+ (-Z),
 !.
```

Similarly, a bound on the quotient of two standard sets is defined by using the multiplicative inverse and the product,

```
X contains Y/Z:-
 X contains (Z inverse)*Y,
 !.
```

In order to bound the `N`–th power (iterate) of an operator `F`, we proceed inductively. By definition, $\mathtt{F}^{\mathtt{0}}$ is the identity,

```
X contains F**0 of X :-
 !.
```

and for positive values of `N` we use the fact that $\mathtt{F}^{\mathtt{N}}(\mathtt{X}) = \mathtt{F}^{\mathtt{N-1}}(\mathtt{F}(\mathtt{X}))$,

```
Y contains F**N of X :-
 M is N-1,
 X1 contains F of X,
 Y contains F**M of X1,
 !.
```

Powers are also used for approximate computations. Thus, we define

```
X approximates F**0 of X :-
 !.
```

and

```
Y approximates F**N of X :-
 M is N-1,
 X1 approximates F of X,
 Y approximates F**M of X1,
 !.
```

13.2. Expression Evaluation

Below we will define two clauses for goals of the form `X contains E`, where `E` is a term containing at least one argument that needs to be evaluated (as a bound) before any of the clauses from previous sections applies. A bound on the identity map for numbers

```
i(X,Y) contains i(X,Y):-
 !.
```

or for polynomials

```
p(L) contains p(L):-
 !.
```

or for functions

```
f(P,E) contains f(P,E):-
 !.
```

will be used for arguments that are already standard sets.

The following clause applies to expressions `E` consisting of a unary operation `F` applied to some other expression `X`:

```
Y contains E:-
 E =..  [F,X],
 E1 =..  [F,S],
 S contains X,
 Y contains E1,
 !.
```

Namely, we first use the predicate `=../2` (see Subsection 2.7) to define a new expression `E1` that is identical to `E`, but with the subexpression `X` replaced by a variable `S`. In the next step, `X` is evaluated (as a bound) and the result is placed in `S`. Now that `S` is a standard set, the goal `Y contains E1` is handled by the clauses from previous sections.

Similarly, in the case of a binary operation we have

```
Y contains E:-
 E =..  [F,X1,X2],
 E1 =..  [F,S1,S2],
 S1 contains X1,
 S2 contains X2,
 Y contains E1,
 !.
```

Here, one of the subexpressions `X1` or `X2` may be a standard set, in which case it is evaluated by using the appropriate (bound for the) identity map defined above.

13.3. Sorting Numbers

If L1 is instantiated to a list of representable numbers, then the following clauses for sort_numbers/2 will make sort_numbers(L1,L2) true, with L2 the list of numbers from L1 sorted in increasing order. We use sorting by insertion. Unless the given list is empty and thus sorted,

```
sort_numbers([],[]):-
 !.
```

we start by sorting the list consisting of all but the first number, and then insert this number in the right place

```
sort_numbers([H1|T1],L):-
 sort_numbers(T1,T2),
 insert_a_number(H1,T2,L),
 !.
```

by going down the already sorted part of the list, if necessary,

```
insert_a_number(X,[H|T],[H|L]):-
 X>H,
 insert_a_number(X,T,L),
 !.
```

until the number can be prepended without being in front of a smaller element

```
insert_a_number(X,L,[X|L]):-
 !.
```

14. Proving the Theorem

In this section we give a proof of Theorem 6.1, up to trivial numerical estimates that can be carried out by a computer. We also define the approximate polynomial solution G_0 and the two numbers ε and ρ mentioned in Theorem 6.1.

Given that the operator $\mathcal{R}$ will be shown to be a contraction in some neighborhood of G_0, it is natural to construct G_0 by iterating an approximate version of $\mathcal{R}$, starting with a rough initial guess for the fixed point. The transformation $\mathcal{R}_{\text{approx}}$ is defined (on standard sets for polynomials) by the clause

```
P1 approximates r of P2:-
 zero_error(E),
 f(P,_) contains r of f(P2,E),
 P1 approximates P,
 !.
```

We are now ready to combine the bounds from previous sections into a proof/0. Starting with an initial Guess containing the function $z \mapsto -0.4 \cdot z$, we iterate $\mathcal{R}_{\text{approx}}$ thirteen times,

```
proof:-
 I contains -i(4,4)/i(10,10),
```

```
Guess=p([i(0,0),I]),
P0 approximates r**13 of Guess,
zero_error(E),
G0=f(P0,E),
```

in order to get a set `G0` whose single element is the desired polynomial G_0. The number ε in Theorem 6.1 is now defined to be the following upper bound `Eps` on the norm of $\mathcal{R}(G_0) - G_0$:

```
i(_,Eps) contains norm((r of G0)-G0),
write('Epsilon is approx.='),write(Eps),nl,
```

Next, we construct a standard set `U` containing the ball $U_\beta(G_0) \subset \mathcal{B}_0$ of radius β centered at G_0,

```
radius_of_ball_in_function_space(Beta),
write('Beta is approx.='),write(Beta),nl,
domain(D),
Ball contains i(0,Beta)/D,
U=f(P0,e(Ball,i(0,0))),
```

and determine an upper bound ρ on the operator norm of the derivative $D\mathcal{R}_G$, for every point in the standard set `U`,

```
i(_,Rho) contains operator_norm(d(r) at U),
write('Rho is approx.='),write(Rho),nl,
```

Finally, after checking the inequality $\varepsilon < (1-\rho) \cdot \beta$,

```
i(Rhs,_) contains (i(1,1)-i(Rho,Rho))*i(Beta,Beta),
Eps < Rhs,
```

the proof of Theorem 6.1 is completed.

```
write('q.e.d.'),nl,
!.
```

With the following headless clause, we ask Prolog to verify the `proof`, by going through the steps indicated above and carrying out the corresponding numerical estimates, as specified in the previous sections:

```
:- proof.
```

15. Appendix: Running the Program

The reader who would like to carry out the `proof` is invited to go through the following steps. (We assume that the reader has access to the file proof.pl containing the program. This file is produced e.g. by running the .tex file of this paper through the TEXcompiler.) After the Prolog system is started, the following prompt will appear:

```
?-
```

At this point, it suffices to type

```
consult('proof.pl').
```

and then to hit the "return" key. This makes Prolog read through the file proof.pl in the way described in Subsection 2.6. When Prolog reaches the clause `:-proof`, it makes an attempt to verify the `proof`. If Prolog succeeds (as it will, if it conforms to the standards described in Sections 2 and 3) then the program will terminate by displaying "q.e.d.", and the prompt `?-` will return.

Acknowledgements

P.W. and H.K. would like to thank the Department of Mathematics at the University of Texas at Austin and the Department of Theoretical Physics at the University of Geneva, respectively, for their hospitality while part of this work was carried out. We are grateful to Logic Programming Associates Ltd (LPA) for donating a copy of their Prolog compiler, and to Brian D. Steel at LPA for many helpful discussions. We also thank Jan Wehr and Giampaolo d'Alessandro for a careful reading of preliminary versions of the manuscript.

References

[C] A. Celletti: *Construction of librational invariant tori in the spin–orbit problem.* Journal of Applied Mathematics and Physics (ZAMP), **45**, 61 (1993).

[CC] A. Celletti and L. Chierchia: *Construction of Analytic KAM Surfaces and Effective Stability Bounds.* Commun. Math. Phys., **118**, 119–161 (1988).

[CM] W.F. Clocksin and C.S. Mellish: *Programming in Prolog.* Third, Revised and Extended Edition. Springer–Verlag (1987).

[E] H. Epstein: *Fixed Points of Composition Operators.* In: Nonlinear Evolution and Chaotic Phenomena, G. Gallavotti and P. Zweifel (editors). NATO ASI Series B: Phys., **176**, 71–100 (1987).

[EKW1] J.-P. Eckmann, H. Koch, and P. Wittwer: *Existence of a Fixed Point of the Doubling Transformation for Area–Preserving Maps of the Plane.* Phys. Rev. A, **26**, 720–722 (1982).

[EKW2] J.-P. Eckmann, H. Koch and P. Wittwer: *A Computer–Assisted Proof of Universality for Area–Preserving Maps.* Memoirs of the American Mathematical Society, **47**, 1–121 (1984).

[EMO] J.-P. Eckmann, A. Malaspinas, and S. Oliffson-Kamphorst: *A software tool for analysis in function spaces.* In: Computer Aided Proofs in Analysis, K. Meyer and D. Schmidt (editors). The IMA Volumes in Mathematics, **28**, 147–166 (1991).

[EW1] J.-P. Eckmann and P. Wittwer: *Computer Methods and Borel Summability Applied to Feigenbaum's Equation.* Lecture Notes in Physics, **227**. Springer–Verlag, Berlin Heidelberg New York (1985).

[EW2] J.-P. Eckmann and P. Wittwer: *A complete proof of the Feigenbaum conjectures.* J. Stat. Phys., **46**, 455–477 (1987).

[FL] C. Feffermann and R. de la Llave: *Relativistic Stability of Matter, I.* Revista Matemática Iberoamericana, **2/1,2**, 119–213 (1986).

[FS] C. Feffermann and L. Seco: *Aperiodicity of the Hamiltonian Flow in the Thomas–Fermi Potential.* Revista Matemática Iberoamericana, **9/3**, 409–551 (1993).

[IEEE] *The IEEE Standard for Binary Floating–Point Arithmetic.* ANSI/IEEE Std 754–1985. (Can be ordered from IEEE Standard Board, 345 East 47th Street, New York, NY 10017, USA).

[KW1] H. Koch and P. Wittwer: *A Non–Gaussian Renormalization Group Fixed Point for Hierarchical Scalar Lattice Field Theories.* Commun. Math. Phys., **106**, 495–532 (1986).

[KW2] H. Koch and P. Wittwer: *Rigorous Computer–Assisted Renormalization Group Analysis.* In: VIIIth International Congress on Mathematical Physics, M. Mebkhout and R. Sénéor (editors). World Scientific (1986).

[KW3] H. Koch and P. Wittwer: *Computing Bounds on Critical Indices.* In: Non–Linear Evolution and Chaotic Phenomena, G. Gallavotti and P. Zweifel (editors). NATO ASI Series B: Phys., **176**, 269–277 (1987).

[KW4] H. Koch and P. Wittwer: *The Unstable Manifold of a Nontrivial RG Fixed Point.* Canadian Mathematical Society, Conference Proceedings, **9**, 99–105 (1988).

[KW5] H. Koch and P. Wittwer: *On the Renormalization Group Transformation for Scalar Hierarchical Models.* Commun. Math. Phys., **138**, 537–568 (1991).

[KW6] H. Koch and P. Wittwer: *A Nontrivial Renormalization Group Fixed Point for the Dyson–Baker Hierarchical Model.* Commun. Math. Phys., **164**, 627–647 (1994).

[KW7] H. Koch and P. Wittwer: *Bounds on the Zeros of a Renormalization Group Fixed Point.* Preprint Austin/Geneva (1994).

[KP] R.S. MacKay and I.C. Percival: *Converse KAM: Theory and Practice.* Commun. Math. Phys., **98**, 469–512 (1985).

[La1] O.E. Lanford: *Remarks on the accumulation of period–doubling bifurcations.* In: Mathematical Problems in Theoretical Physics. Lecture Notes in Physics, **116**, 340–342 (1979).

[La2] O.E. Lanford: *Computer–Assisted Proof of the Feigenbaum Conjectures.* Bull. A.M.S. (New Series), **6**, 427–434 (1982).

[La3] O.E. Lanford: *Computer–Assisted Proofs in Analysis.* Physica, **124 A**, 465–470 (1984).

[La4] O.E. Lanford: *A Shorter Proof of the Existence of the Feigenbaum Fixed Point.* Commun. Math. Phys., **96**, 521–538 (1984).

[Lla] R. de la Llave: *Computer assisted proofs of stability of matter.* In: Computer Aided Proofs in Analysis, K. Meyer and D. Schmidt (editors). The IMA Volumes in Mathematics, **28**, 116–126 (1991).

[Llo] J.W. Lloyd: *Foundations of Logic Programming.* Second, Extended Edition. Springer–Verlag (1987).

[LPA] Logic Programming Associates Ltd., Studio 4, The Royal Victoria Patriotic Building, Trinity Road, London SW18 3SX, England.

[LR1] R. de la Llave and D. Rana: *Accurate Bounds in K.A.M. Theory.* In: VIIIth International Congress on Mathematical Physics, M. Mebkhout and R. Sénéor (editors). World Scientific (1986).

[LR2] R. de la Llave and D. Rana: *Accurate Strategies for Small Denominator Problems.* Bull. Amer. Math. Soc., **22**, 85-0-90 (1990).

[LR3] R. de la Llave and D. Rana: *Accurate strategies in K.A.M. problems and their implementation.* In: Computer Aided Proofs in Analysis, K. Meyer and D. Schmidt (editors). The IMA Volumes in Mathematics, **28**, 127–146 (1991).

[M] B.D. Mestel: *A computer assisted proof of universality for cubic critical maps of the circle with golden mean rotation number.* Ph.D. Thesis, Math. Dept., University of Warwick (1985).

[Ra] D. Rana: *Proof of Accurate Upper and Lower Bounds for Stability Domains in Small Denominator Problems.* Ph.D. Thesis, Princeton University (1989).

[Ro] P. Ross: *Advanced Prolog.* Techniques and Examples. Addison–Wesley Publishing Company (1989).

[Se1] L. Seco: *A Lower Bounds for the Ground State Energy of Atoms.* Thesis, Princeton University (1989).

[Se2] L. Seco: *Computer Assisted Lower Bounds for Atomic Energies.* In: Computer Aided Proofs in Analysis, K. Meyer and D. Schmidt (editors). The IMA Volumes in Mathematics, **28**, 241–251 (1991).

[SP] Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden.

[SS] L. Sterling and E. Shapiro: *The Art of Prolog.* Advanced Programming Techniques. The Massachusetts Institute of Technology (1988).

[St] A. Stirnemann: *Existence of the Sigal Disc Renormalization Fixed Point.* Nonlinearity, **7**, 959–974 (1994).

[XSC] E. Adams and E. Kulisch (editors): *Scientific Computing with Automatic Result Verification.* Applications in the Engineering Sciences. Academic Press, San Diego (1993).