Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfasst habe, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch ansonsten keiner unerlaubten Hilfe bedient habe.

Linz, Juni 2003

Dipl.-Ing. Markus Rosenkranz

Zusammenfassung

Obwohl Randwertprobleme (RWP) wohl zu den wichtigsten Problemtypen aus Physik, Chemie oder auch Finanzwissenschaft gehören, ist ihre Behandling im Symbolic Computation noch ziemlich dürftig. Bislang werden noch die einfachsten RWP gewöhnlich entweder numerisch oder per Hand gelöst, allenfalls auf ad-hoc Basis unterstützt durch Computeralgebra-Pakete. Wohl hat man im Symbolic Computation diverse Werkzeuge zum Lösen von Differentialgleichungen, aber ihre Anwendung auf RWP ist weitgehend unbefriedigend.

In dieser Dissertation legen wir eine neue Methode zur Lösung von RWP für lineare gewöhnliche Differentialgleichungen mit konstanten Koeffizienten vor. Im Unterschied zu den bisherigen Methoden, die vermittels der Greenschen Funktion alles auf die Funktionsebene zurückführen, fungiert unser Verfahren ganz auf Operatorebene. Die nötigen Operatoren werden in unserer Methode als nichtkommutative Polynome repräsentiert, wobei Basisoperatoren wie Differentation, Integration und Randauswertung als Unbestimmte auftreten.

Der entscheidende Schritt zur Lösung des RWP besteht darin, den gesuchten Greenschen Operator als eine geeignete Moore-Penrose-Schiefinverse aufzufassen. Um die daraus hervorgehenden Gleichungen nach dem Greenschen Operator aufzulösen, wird eine sorgfältig zusammengestellte nichtkommutative Gröbnerbasis verwendet, welche die wesentlichen Interaktionen zwischen den Basisoperatoren widerspiegelt.

Wir haben unser Verfahren implementiert als *Mathematica* Paket, eingebettet in das in B. Buchbergers Gruppe entwickelte *Theorema* System. Ein Teil der Dissertation kann auch als Bedienungsanleitung für diese Implementierung gelten.

Abstract

Although boundary value problems (BVPs) are among the most important problem types coming from physics, chemistry and even finance, their coverage in symbolic computation is still rather weak. Up to now, even the simplest BVPs are usually solved either numerically or by some handcrafted calculations, possibly supported by some computer algebra package in an ad-hoc way. Symbolic computation does have several tools for solving differential equations, but their application to BVPs is largely unsatisfactory.

In this thesis, we present a new method for solving BVPs for linear ordinary differential equations with constant coefficients. Unlike existing methods that reduce everything to the functional level via the Green's function, our approach works on the level of operators throughout. Our method proceeds by representing the operators needed as noncommutative polynomials using as indeterminates basic operators like differentation, integration, and boundary evaluation.

The crucial step for solving the BVP is to understand the desired Green's operator as a suitable oblique Moore-Penrose inverse. The resulting equations are then solved for the Green's operator using a carefully compiled noncommutative Gröbner basis that reflects the essential interactions between the basic operators.

We have implemented our method as a *Mathematica* package embedded into the *Theorema* system developed in B. Buchberger's group. Part of the thesis may also be regarded as a user's manual for this implementation.

Table of Contents

Preface 6

Prolog: Genesis of the Green's Polynomials 8

The Algebra of Green's Polynomials 10

The Example of Steady Heat Conduction in a Rod 10
The Green's Polynomials 22
Inverting Differential Operators 47
Computing the Nullspace Projector 49
Finding the Green's Operator 52

A User's Manual for the Green's Package 56

The Theorema Environment 56
The Overall Design of the Green's Suite 67
The Reductor for Noncommutative Polynomials 70
The Matrix Evaluator 87
The Green's Evaluator 91

Implementation Notes 94

General Design Principles in Theorema Programming 94
Organization of the Green's Suite 96
Implementation of the Polynomial Reductor 97
Implementation of the Matrix Evaluator 103
Implementation of the Green's Evaluator 104

Appendix: The Concept of Polynomial 107

A Sloppy Definition 107
A Rigorous Definition 109
An Alternative and More General Definition 111
Computing with Polynomials 125
Commutative versus Noncommutative 128

Epilog: Prospects of Generalization 132

Curriculum Vitae 134

Acknowledgements 136

References 139

Preface

The present PhD thesis has grown out of a cooperation between the group of B. Buchberger and the group of Heinz W. Engl at the Johannes Kepler University of Linz; see the Prolog for a short overview of its genesis. Its main thrust is to extract the algorithmic power contained in the concept of Moore-Penrose inverse. This line of thought is also reflected in the *outline of its contents*:

Chapter 1 constitutes the theoretical basis of the thesis. It develops the crucial concept of the *Green's polynomials* (together with their algorithmic incarnation: the system of Green's identities), following the idea of representing the Green's operator of a BVP as a suitable oblique Moore-Penrose inverse. Using this methodology, everything is reduced to the problem of determining a suitable nullspace projector for the Moore-Penrose inverse and a right inverse of the given differential operator. These two subproblems are readily solved by two concise formulae, thus turning the whole approach into an overall algorithm for solving BVPs. The chapter concludes with the correctness proof of this algorithm.

We have also implemented our algorithm as a *Mathematica* package, embedded into the *Theorema* system developed in B. Buchberger's group. Chapter 2 is a *user's manual* for this implementation, which we have called the Green's suite. After a short overview of the general *Theorema* environment and the global setup of our package, we describe its three main components: the reductor for noncommutative polynomials, the matrix evaluator, and the Green's evaluator.

We round up with a concise description of the most important *implementation issues* in Chapter 3. Unlike the previous chapter, which is explicitly written for a user of our system, the last chapter is intended to provide some background material potentially valuable for a programmer building on our implementation of the Green's suite. We follow the same structure as in the previous chapter, describing first a couple of general issues related to *Theorema* programming, then some overall aspects of how the Green's suite is organized, and finally implementation details of the polynomial reductor, the matrix evaluator, and the Green's evaluator.

Since noncommutative polynomials are the main fabric from which we have woven the relevant structures, the interested reader might be curious how such these objects can be introduced in a rigorous manner—following a uniform paradigm. We have therefore prepared a fairly comprehensive appendix on the *concept of polynomial*, written from a dedicated logical viewpoint. In our formulation, we use the notion of categories and functors in the sense introduced by B. Buchberger for automated theorem proving and computer-supported formalization. It turns out that such a view establishes a lucid connection between the general polynomial concept and Gödel's completeness proof—a synopsis of two themes not often seen together. And a beautiful dejà-vu of some topics that got me started in B. Buchberger's group (and I hope this dejà-vu will not be the last of its type).

The thesis is framed by a Prolog sketching the interesting *genesis* of its main ideas and an Epilog suggesting various lines of *future research*.

Before going into the actual subject matter, we should fix a few issues of convention and notation. First of all, the reader may have noticed that we use the *definite article* with words like "Green's function", "Green's operator" or "Green's polynomials". In the literature, there seems to be some disagreement on whether one should say just "Green's function" or "the Green's function". For example, [63] uses the first variant, whereas [45] uses the second. But as soon as an adjective is adjoined to these terms, the definite article is unavoidable anyway (for both authors), as in the phrase "the modified Green's function". Therefore we have decided to use the definite article throughout.

Regarding *single and double quotes*, we tried to be consistent with the following convention: We use double quotes of actually quoting a phrase, whereas single quotes are reserved for relativizing something (meaning that one 'abuses' a certain phrase as a kind of metaphor or one is not 'deadly serious' about it).

The numbering scheme used in *formal units* like "Definition" or "Lemma" was only necessary in Chapter 1, where we have numbered everything in one sequence. All formal units are ended by a □ sign. The formal unit "Input" is used for *Theorema* input that is sent verbatim to the *Mathematica* kernel: so if you have the text as a notebook rather than on paper, you can evaluate these cells by selecting their cell brackets and pressing SHIFT-ENTER. Analogously, the formal unit "Computation" represents *Theorema* computations carried by the *Mathematica* kernel. The text above the horizontal line is verbatim input; the text below verbatim output.

About formal issues we need not say much since we will work in a very normal setting, using only standard notation unless explained otherwise. Thus the reader may assume first-order predicate logic with Zermelo-Fraenkel set theory as the underlying *foundation system* (possibly enhanced by logical sorts as in Section 3 of the Appendix). Free variables are understood as universally quantified. Let Φ be a formula and T a term (typically containing x as a free variable). Then the notation $\Phi_{x\leftarrow T}$ denotes Φ with all free occurrences of the variable x replaced by the term T. Furthermore, we use $x \mapsto T$ for the lambda quantifier on T, usually expressed as $\lambda_x T$ or $\lambda x . T$ in computer science texts.

The set of *natural numbers* is understood to include zero and is denoted by $\mathbb{N}$; the positive natural numbers are denoted by $\mathbb{N}^\times$ (if A is any monoid, its unit group $A^\times$ is given by the set of all its invertible elements). Unless otherwise specified, the *range* of k, l, m, n is $\mathbb{N}$, that of x, y, z is $\mathbb{R}$.

Following [46], we write $f^*(X)$ and $f_*(X)$ for denoting the direct and inverse image of a set X under a function f, respectively. As usual in mathematics—especially in operator theory—we will sometimes omit the parentheses used for function application when the context excludes ambiguity.

Unless otherwise specified, a *vector space* (in particular, an algebra) $\mathcal{A}$ is assumed to have $\mathbb{C}$ as its field of scalars. Accordingly, we mean "$\mathbb{C}$-linear" when we say "linear".

Moreover, we will always conceive a vector space $\mathcal{A}$ as containing a specific *basis*, which will be denoted by $\mathcal{A}^\#$. In formal terms, this means the following: Whereas a complex vector space is usually understood as a carrier set V with vector addition $+ : V \times V \to V$, the zero vector $0 \in V$, the negative vector $- : V \to V$, and scalar multiplication $\cdot : \mathbb{C} \times V \to V$, we add to this signature an additional operation $^\# : \mathbb{N} \to V$, yielding the chosen sequence of basis vectors. (Strictly speaking, we should call $\mathcal{A}$ a "based vector space" or a "based algebra", but we will refrain from doing so in order to keep the language simple.)

The *transpose* of a matrix A is written as $A^\top$.

The *reflexive-transitive closure* of $\to$ is denoted by $\overset{*}{\to}$, the *symmetric* one by $\leftrightarrow$, the *reflexive-symmetric-transitive* one by $\overset{*}{\leftrightarrow}$.

Prolog: Genesis of the Green's Polynomials

The ideas presented in this PhD thesis emerged in the stimulating atmosphere of the so-called Hilbert seminars held in Linz and Hagenberg between October 2001 and June 2002. These seminars were organized by B. Buchberger and Heinz W. Engl in the frame of a *special research project* (in German "Spezialforschungsbereich", abbreviated SFB) founded in 1998 at the Johannes Kepler University of Linz. This project—associated with the FWF grant F13—joins several institutes with the explicit goal of establishing bridges between "symbolic mathematics" and "numeric mathematics", two firmly established key disciplines of modern mathematics—both highly algorithmic in spirit and yet so different in character. It is the conviction of this SFB that both disciplines contain a rich potential of interaction, and this vision has already triggered quite some interesting cooperation projects between various symbolic and numeric groups at the Johannes Kepler University.

Embedded into this context, the *Hilbert seminars* served as a unique instrument for establishing a fundamentally new link between symbolic methodology and numeric heritage; it is the object of this Prolog to describe the nature and genesis of this link.

The idea of the seminars was to explore the topic of *operator theory* both from a symbolic and numeric viewpoint. For this purpose, the two principal investigators started with some fundamental lectures providing the necessary background as seen from their respective field—Heinz W. Engl from the numeric side and B. Buchberger from the symbolic side.

On the numeric side, Heinz W. *Engl* gave a series of lectures about the following topics:

- *Ill-posed problems* and their intrinsic numeric difficulties.
- The usage of *Tikhanov regularization* for mastering these difficulties.
- The *Moore-Penrose inverse* as a convenient conceptual frame for "solving ill-posed problems".
- Regularization in the light of s*ingular value decomposition.*

On the symbolic side, B. *Buchberger* focused on the following points in his lectures:

- The necessity of a *rigorous formal language* for representing problems in the sense of symbolic computation.
- The role of *predicate logic* as the universal linguistic frame for both symbolic and numeric mathematics.
- The usage of *computer-supported tools* like *Theorema* once a problem has a rigorous formulation.

After these initial lectures, the groups focused on the following *two relevant papers*. Heinz W. Engl suggested the study of [27], exploring a the concept of the oblique Moore-Penrose inverse from a minimization viewpoint. After a presentation of the main points of this paper, delivered by Benjamin Hackl on January 23 in 2002, the group discussed potential connections to symbolic methods. The other paper [34], together with the companion papers [67] and [35], was suggested by B. Buchberger and presented by Teimuraz Kutsia also on January 23; it discusses the usage of noncommutative Gröbner bases for simplifying operator expressions arising in control theory.

Both of these papers contained some *crucial ingredients* preparing the results exposed in this PhD thesis:

- The first paper opened the view for the *oblique Moore-Penrose inverse* and its additional freedom available in it by the choice of nullspace and range projectors. This particular concept of generalized inverse also provided a convenient equational characterization that is in a sense more fundamental than the more common standard Moore-Penrose inverse in Hilbert spaces (corresponding to orthogonal projectors), where the projector equalities are usually replaced by the more abstract self-adjointness conditions for the square operators.
- The second paper showed a simple way of applying *noncommutative Gröbner bases for manipulating operators*. However, they do not consider solving any operator equations, they are only interested in transforming compound operators (often covering several pages) into simpler forms by applying certain given operator equalities. Since Gröbner bases were discovered by B. Buchberger in his PhD thesis [17] (see also the journal version [7] and the survey article [13]), this line of research was naturally all the more attractive.

The fundamental insight needed for putting these two things together is the following *chain of thoughts*: It is well known that noncommutative Gröbner bases—just as their commutative companions—may not only be used for simplifying terms but also for solving equations. The simplest equation on the operator level is a BVP, so maybe we could use noncommutative Gröbner bases on them? But usually one sees a BVP as an inversion problem for a differential operator that is made bijective by incorporating the boundary conditions into its domain—but this elegant trick does not have any immediate computational content. Therefore the crucial idea is to remove the boundary conditions from the artificial domain of the differential operator and search for a Moore-Penrose inverse instead (see the explanation after Equation 2 in Chapter 1).

It turned out that this scheme *actually works*: As we show in detail in Chapter 1, the additional freedom available in the choice of the projectors for the Moore-Penrose inverse is always sufficient for taking care of the boundary conditions. Various basic operations like differentiation, integration and boundary evaluation are represented by polynomial indeterminates. The Moore-Penrose equations can be augmented by suitable noncommutative polynomial equalities for describing the algebraic essence of the various relations between the basic operations. Finding the Green's operator corresponds to solving the resulting noncommutative polynomial system. This is essentially the procedure presented by the author in the Hilbert Seminar on March 20, 2002; in a polished form, it is published in the journal paper [57].

In the course of further research, *extensive simplifications* to the strategy described above were found. In particular, it turned out that only one of the four Moore-Penrose equations is actually needed, and the expensive computation of noncommutative Gröbner bases can be avoided using a carefully selected precomputed one. This new approach was presented in a poster at ISSAC'2003, to be published as an extended abstract in [60]. In the present PhD thesis, it is for the first time described in full detail—the key concept is a certain noncommutative polynomial algebra, which we have called the Green's algebra—see Proposition 31 in Chapter 1.

1 The Algebra of Green's Polynomials

In this chapter, we will present the core results of the PhD thesis—a new method for solving regular BVPs for linear differential equations with constant coefficients (see Definition 7 for the precise specification). We will first motivate our idea in Section 1 by the simple example of steady heat conduction in a rod. Following these traces, Section 2 develops the main tools used for attacking BVPs in our sense—most importantly, the system of Green's equalities given in Input 14 and their associated algebra of Green's polynomials introduced in Proposition 31. With these tools, it is possible to design a complete algorithm for solving BVPs: The first step is to right-invert the given differential operators as described in Section 3. The right inverse so obtained is then multiplied by a certain nullspace projector, whose computation is discussed in Section 4. Finally, we put all things together in Section 5, concluding the chapter with the correctness proof for the overall algorithm of solving BVPs.

1.1 The Example of Steady Heat Conduction in a Rod

Before going to the relevant definitions and theorems, let us give a 'classical' example of what type of problems we have in mind and how we go about to solve them. In this chapter, we are concerned with BVPs for ordinary linear differential operators with constant coefficients (we will henceforth refer to such problems simply as BVPs). One of the simplest and yet non-trivial problems of this type is exemplified by the one-dimensional heat equation with constant conductivity and fixed temperature at both ends. One sees this example or some minor variant in almost any introductory chapter on linear BVPs; see e. g. page 42 in [63] or page 13 in [33]. Using suitable units, this BVP can be formulated thus:

Given $f \in C[0, 1]$,

find $u \in C^2[0, 1]$

such that

$$\begin{aligned} &u'' = f\,, \\ &u(0) = u(1) = 0\,. \end{aligned} \tag{1}$$

The *proper topological setting* for this problem is to consider both $C[0, 1]$ and $C^2[0, 1]$ as Banach spaces with the Chebyshev norm $\|\cdot\|_\infty$. The differentiation operator $D : u \mapsto u'$ is then conceived as a *partial* function on $C[0, 1]$ with dense domain $C^1[0, 1]$. For the example above, this means that D^2 operates on $C[0, 1]$ with dense domain $C^2[0, 1]$. Note that D^2 is a closed and *discontinuous* operator in this view, but this is all one needs for applying the crucial Propositions 2 and 3 cited below. One could think it is preferable to take $C^2[0, 1]$ with its canonical norm: Then the operator D^2 would be *total,* and it would even be *continuous*, not just closed. But the drawback of this setting is that $C^2[0, 1]$ is not the 'natural' domain for the operator D^2 as it is usually used in Sturm-Liouville theory. If the elements of $C^2[0, 1]$ are obtained from measurements, the data error is typically estimated in the norm of $C[0, 1]$, and it would not be reasonable to ask for estimates in the norm of $C^2[0, 1]$, as this would involve the unstable process of differentiating data; see [26] for details.

For our purposes, though, we can us *ignore all topological notions* by assuming that f and hence u are in $C^\infty[0, 1]$, which we shall view as a "naked" vector space without any topology. Of course, allowing only smooth right-hand sides f is rather restrictive: as explained above, the natural assumption would be that f is just continuous. But actually even this is too restricted for practical purposes, where one typically deals with weak solutions living in various Sobolev spaces. And if one wants to include those, it turns out to be more efficient to change over to the completely distributional setting $C_0^{-\infty}[0, 1]$ subsuming all the Sobolev spaces one can ask for; we will do this in Definition 37. In fact, both $C^\infty[0, 1]$ and $C_0^{-\infty}[0, 1]$ have a slightly unusual topology: they are locally convex vector spaces that cannot be made into Banach spaces in any natural way. For the purpose of solving BVPs symbolically, however, these topologies are not relevant.

Of course one may always prove topology-related statements in an independent effort. For example, it is not difficult to show *within* the distributional setting that restricting f to $C[0, 1]$ leads to u always being in $C^2[0, 1]$; this corresponds to the classical solution concept described above. The point we want to make here is that distributions allow us to *separate the topological side from the algebraic one*, and we want to focus on the latter. The only topology we need here is for defining differentiation and integration; but having done so, we can immediately "forget" the topology and view these operators as plain linear transformations as explained above.

Using the smooth setting, all the operators involved will have the type $C^\infty[0, 1] \to C^\infty[0, 1]$. Besides this, we note that the BVP (1) is regular in the sense that it has precisely one solution (throughout this thesis we will deal only with regular problems). Now solving a problem like (1) means that one can assign a solution $u \in C^\infty[0, 1]$ to each so-called forcing function $f \in C^\infty[0, 1]$. In other words, the solution "is" an operator $G : C^\infty[0, 1] \to C^\infty[0, 1]$, usually named the *Green's operator*. But not only the solution, also the relevant data for this problem are encoded in operators:

- The *differential equation* $u'' = f$ is formed by the operator $D^2 = u \mapsto u''$. The forcing function f occurring on its right-hand, however, does not contain any significant information—it serves only as a placeholder for the functional formulation given in (1). In operator-theoretic terms, the differential equation is simply $D^2 G = I$; applied to an arbitrary function $f \in C^\infty[0, 1]$, this givens (1) because $G f \in C^\infty[0, 1]$ is the solution u by the very definition of G. We use I for the identity operator on whatever space is considered; here this is of course $C^\infty[0, 1]$.

- The *boundary conditions* $u(0) = 0$ and $u(1) = 0$ can be formulated by the corresponding boundary operators. On the left-hand side, this is the left-boundary operator L mapping u to the constant function $x \mapsto u(0)$; on the right-hand side, it is the corresponding operator R mapping u to $x \mapsto u(1)$. Using these operators, the boundary conditions can now be written on the operator level as $L G = O$ and $R G = O$; the functional formulation follows from this as before. Here we have written O for the null operator mapping every function of $C^\infty[0, 1]$ to the null function $x \mapsto 0$.

So the natural interpretation of (1) is clearly situated *on the operator level*: Given the differential operator D^2 and the boundary operators L and R, find the Green's operator G for them, i.e. find G such that

$$\begin{aligned} &D^2\, G = I, \\ &L\, G = O, \\ &R\, G = O \end{aligned} \tag{2}$$

In other words, G should be a *right inverse* of D^2, and it should be *annihilated* by L and R. Whereas there are many right inverse of D^2, the annihilation constraints are supposed to single out one particular right inverse—which must then be the uniquely defined Green's operator. (Observe that we could also view D^2 as an operator from $\{u \in C^\infty[0, 1] \mid u(0) = u(1) = 0\}$ to $C^\infty[0, 1]$. In this case, it is of course invertible, and G is its two-sided rather than just right inverse. But the problem with this approach is that it swallows the information contained in the two boundary equations into the domain specification of D, where we lack any natural computational access. Therefore such a model is more useful for proving abstract properties about the problem than for finding its concrete solution.)

The main challenge in the equation system (2) is that it does not give us easy access to the unknown G. So it may be a good idea to have a look at some other equations analysis offers us for characterizing 'almost inverse' operators like the right inverse G in our case. Now there is one famous concept in analysis, which generalizes operator inversion—the *Moore-Penrose inverse* [72]. Any(!) linear operator between two inner product spaces (in the topological setting: any bounded linear operator between two Hilbert spaces) has a uniquely determined Moore-Penrose inverse, which is as close to an inverse as possible in the following sense: It maps each point back to a location with minimal distance to the original point; and among all such locations, it selects the one with minimal norm. Hence one can view the Moore-Penrose inverse as a two-stage minimizer, working first on the codomain and then on the domain.

The concept of the Moore-Penrose inverse, though sufficiently flexible for many purposes, is still not good enough for us: We need a very specific right inverse fulfilling some particular boundary conditions, and chances are that it does not coincide with the unique Moore-Penrose inverse; in fact, this suspicion will be confirmed soon. So we need an even more general concept of inverse that allows some freedom for possibly incorporating the boundary requirements. Such a concept exists, and it is rightly called the *generalized inverse*; see pages 14-16 in [72]. The idea is that we may change the distance measure both for the codomain and the domain minimization. Whereas the original minimizers are realized by orthogonal projections, the modified ones use oblique projections (generalized inverses in Hilbert spaces are therefore also called oblique inverses). More generally, one can introduce generalized inverses in plain vector spaces (in the topological setting typically: Banach spaces [27]), because projection makes sense even when there is no inner product and hence no concept of angle and orthogonality. So the Moore-Penrose inverse is just the 'canonical' generalized inverse corresponding to choosing both projectors to be orthogonal.

1 Definition (Generalized Inverse in Vector Spaces)

Let X and Y be vector spaces, and let T be a linear operator from X to Y. Choose projectors P and Q onto $N = \mathcal{N}(T)$ and $R = \mathcal{R}(T)$, respectively, and let M and S be the corresponding complements $(I - P)_* X$ and $(I - Q)_* Y$. Then the *generalized inverse* of T relative to these projectors, denoted by $T^\dagger_{P,Q}$, is defined as the linear extension of $(T|_M)^{-1}$ with nullspace S.

□

2 Proposition (Elementary Properties of Generalized Inverses in Vector Spaces)

Using the above notation and assumptions, $T^\dagger_{P,Q}$ is a uniquely defined linear operator from Y to X with *nullspace* S and *range* M.

□

3 Proposition (Moore-Penrose Equations in Vector Spaces)

Again using the above notation and assumptions, $T^{\dagger}_{P,Q}$ is uniquely characterized as that linear operator $T^{\dagger}$ from Y to X which fulfills the so-called *Moore-Penrose equations*

$$\begin{aligned} &T\,T^{\dagger}\,T = T, \\ &T^{\dagger}\,T\,T^{\dagger} = T^{\dagger}, \\ &T^{\dagger}\,T = I - P, \\ &T\,T^{\dagger} = Q. \end{aligned} \tag{3}$$

□

Applied to the BVP (2), this means that we try to obtain G as a *generalized inverse* $(D^2)^{\dagger}_{P,Q}$ *for some projectors yet to be determined*. We will therefore write G for this generalized inverse and subsequently show that it is indeed the Green's operator for the given BVP.

Before going on with finding G as a generalized inverse, let us also mention that there is an alternative approach that can somehow be considered the "standard method" of solving BVPs of the type we study here. It proceeds by translating the operator problem into a functional setting in the following way. One can prove that every solution u of a regular BVP for a linear differential operator can be represented as

$$u(x) = \int_0^1 g(x, \xi)\, f(\xi)\, d\xi, \tag{4}$$

where g is called the *Green's function* of the BVP; see e.g. page 189 in [38]. In other words, the Green's operator G is written as an integral operator having the Green's function g as its kernel. Instead of searching for the *operator* G, one can thus search for the (binary!) *function* g. In fact, there are methods for finding g for any regular BVP, using some linear algebra on the fundamental system of the homogeneous differential equation; see for example [63], [37], [38]. The common feature of all these methods is that they immediately work on the level of functions rather than operators—which is not the "natural" setting of this problem, as we pointed out above.

The method we want to present here is, to our best knowledge, a new one. It is genuinely *based on operators* in the sense that it directly computes the Green's operator G rather than the Green's function g, which can be extracted in a trivial post-processing step—if at all desired. For BVPs more complicated than the ones considered here, it is not clear whether such an extraction would be possible; e.g. for nonlinear problems it will typically not be possible. Besides its conceptual superiority, our new method might also involve a gain in efficiency, since we solve a linear system only with numeric rather than functional entries as done in the methods mentioned above.

Now let us go on with the computation of the appropriate generalized inverse for the operator D^2. Its *range* is all of $C^{\infty}[0, 1]$, so Q is the identity I. Now for the nullspace, which is clearly

$$N = \{x \mapsto \alpha x + \beta \mid \alpha, \beta \in \mathbb{R}\} \tag{5}$$

in our example (for a more general case, the determination of the nullspace involves solving a homogenous differential equation). We will try to construct a projector P onto N such that the boundary conditions are also incorporated.

We should note some general features here: On the one hand, any linear differential operator is surjective on $C^{\infty}[0, 1]$, so the range projector Q will *always* be trivial. On the other hand, it will

always be non-injective, so the *nullspace projector* P can *never* be trivial. This is to our best advantage as we would have no chance for the boundary conditions otherwise! In the case of (5), we see that choosing P amounts to specifying for each $u \in C^\infty[0, 1]$ real numbers α, β such that $(P\,u)\,(x) = \alpha\,x + \beta$ for all $x \in [0, 1]$. From Proposition 2 we know that

$$\mathcal{R}(G) = (I - P)_*\, C^\infty[0, 1]. \tag{6}$$

We want to have $u(0) = 0$ and $u(1) = 0$ for all $u \in \mathcal{R}(G)$, so the counter-projections $(1 - P)\,v = v - P\,v$ should vanish at the boundary for all $v \in C^\infty[0, 1]$. Hence we must construct a projector P such that $P\,v$ coincides with v for all $v \in C^\infty[0, 1]$. Now this is a trivial *interpolation problem*: Given a function $v \in C^\infty[0, 1]$, find a linear function $P\,v$ that agrees with v at the grid points $0, 1 \in [0, 1]$. A short calculation leads to

$$\mathrm{P\,u = x \mapsto (1 - x)\,u\,(0) + x\,u\,(1).} \tag{7}$$

We note again that we expect a similar, though more complicated, interpolation problem for any other linear differential operator.

Since we are driving at an *operator-theoretic formulation*, we prefer to define P purely in terms of operators. This can easily be done by introducing the operator X for multiplying with the independent variable, defined by

$$\mathrm{X\,u = x \mapsto x\,u\,(x)} \tag{8}$$

for all $u \in C^\infty[0, 1]$. Using X, the operator P can be characterized as

$$\mathrm{P = (1 - X)\,L + X\,R.} \tag{9}$$

By Proposition 3, G fulfills the Moore-Penrose equations (3); in particular, it is a right inverse of D^2 according to the fourth equation. Furthermore, G fulfills the boundary conditions due to our construction of P. On the other hand, we mentioned above that the Green's operator for a BVP like (1) is uniquely determined, so it must indeed coincide with G as claimed above. Finally we can now apply the characterization result of Proposition 3 in the other direction, yielding the fact that G is uniquely determined by the equations

$$\begin{aligned}
&\mathrm{D^2\,G\,D^2 - D^2 = O,}\\
&\mathrm{G\,D^2\,G - G = O,}\\
&\mathrm{G\,D^2 + (I - X)\,L + X\,R - I = O,}\\
&\mathrm{D^2\,G - I = O,}
\end{aligned} \tag{10}$$

obtained from (3) by substituting the specific operators P, Q, J, K obtained by our considerations. We call (10) the *concrete Moore-Penrose equations* for the BVP (1). Obviously, the first and second equation are redundant, since they follow from the fourth one (In fact, the first redundancy is true for any generalized inverse as observed on page 17 in [72]. The second redundancy, however, is due to the special case of $Q = I$ occurring when the operator to be pseudo-inverted is a linear differential operator with constant coefficients.)

In order to compute with operators, one needs some algebraization for them. Now the obvious data structure to use here is of course the noncommutative polynomial ring $\mathbb{C}\langle G, D, X, L, R\rangle$ because polynomial manipulation is one of the driving forces of computer algebra as pointed out in Section 1 of the Appendix. In fact, all the equations of (10) can be understood very naturally as *polynomial equations* exactly in this sense since we can replace the identity operator I and the null

operator O by the one-polynomial 1 and by the zero-polynomial 0, respectively (as $\mathbb{C}$ is canonically embedded into $\mathbb{C}\langle G, D, X, L, R\rangle$, one usually identifies the complex numbers with the corresponding constant polynomials).

One remark about the scalar field $\mathbb{C}$ is in order here: In practical computation, we will of course work in some *computable subfield* like $\mathbb{Q}$ or in some finite (algebraic and/or transcendental) extension of $\mathbb{Q}$. For theoretical purposes, though, it is more convenient to formulate all the results for the 'mother field' $\mathbb{C}$.

Interpreting now everything in the noncommutative polynomial ring $\mathbb{C}\langle G, D, X, L, R\rangle$, we obtain the following formulation

$$\begin{aligned} &\mathrm{G}\,\mathrm{D}^2 + (1 - \mathrm{X})\,\mathrm{L} + \mathrm{X}\,\mathrm{R} - 1 = 0, \\ &\mathrm{D}^2\,\mathrm{G} - 1 = 0 \end{aligned} \tag{11}$$

for the BVP (1).

Note that from now on, the "operators" G, D, X, L, R are just polynomial indeterminates, carrying no interpretation whatsoever on them, as expounded in Section 3 of the Appendix. So they are merely symbols, knowing nothing about differentiation or boundary values—in particular, they are ignorant of any topology. Hence we must introduce all their 'essential' features gently by adding suitable *interaction equalities* that describe the relevant relations between the operators represented by the various indeterminates.

Besides this, we cannot expect a solution for G when working in $\mathbb{C}\langle G, D, X, L, R\rangle$ because such a solution would be a polynomial in D, X, L, R. But we cannot represent the Green's operator using only differentiation, multiplication and boundary values! In fact, since G is some right inverse of D^2, so to say its 'opposite', it would be natural if it does *not* contain a derivative but rather its opposite—the antiderivative or integral. In other words, we expect G to come out as an integration operator, and it should have a suitable Green's function g as its kernel. Hence we must explicitly add an indeterminate, say A, for representing the *antiderivative*. From the theory of Green's functions we know that their highest derivatives always have a jump on the diagonal; see page 194 in [63]. Typically, $g(x, \xi)$ is given by case distinction as $g_<(x, \xi)$ for the lower triangle $x \le \xi$ and $g_>(x, \xi)$ for the upper triangle $x \ge \xi$ with suitable functions $g_<$ and $g_>$, so the integral representing Green's operator naturally splits into two parts: one going from the left boundary to the diagonal, the other going from the diagonal to the right boundary. So the corresponding operators are

$$\begin{aligned} &\mathrm{A}\,\mathrm{u} = \mathrm{x} \mapsto \int_0^{\mathrm{x}} \mathrm{u}(\xi)\, d\xi, \\ &\mathrm{B}\,\mathrm{u} = \mathrm{x} \mapsto \int_{\mathrm{x}}^1 \mathrm{u}(\xi)\, d\xi, \end{aligned} \tag{12}$$

and we will call them the *integral and cointegral* operators, respectively. Note that $A\,u$ is the antiderivative of u with integration constant chosen such that it vanishes at the left boundary. The cointegral $B\,u$ of u is similar, but the integration constant is chosen such that it vanishes at the right boundary, and the sign is inverted. The operators A and B are duals of each other.

Having these operators available, we should have a good chance of representing all Green's operators with polynomial functions. Hence we will adjoin them as new indeterminates, thus working in the noncommutative polynomial ring $\mathbb{C}\langle G, D, X, A, B, L, R\rangle$.

Now we must set up good interaction equalities. Viewing all of these equalities as oriented from left to right, they form a *reduction system* for simplifying a compound operator involving various mixed occurrences of D, X, A, B, L, R. This suggests that we should think of an intuitive strategy that brings such compound operators more and more towards a canonical form. As we can

always expand polynomials into a sum of monomials, it will suffice to think about what we do to the words over the alphabet $\{G, D, X, A, B, L, R\}$. Let us try to do this as systematically as possible.

In the books, one always sees differential operators in a form where the D^k is moved to the outmost right position. This suggests that we should first study those 'interactions' where the operator D is on the left and some other operator is on the right, and that we should try to 'move' D across the operator on the right. If this operator is X, we can obviously achieve this goal by virtue of the *product rule of differentiation*, which is

$$\mathrm{D}\,\mathrm{X} = \mathrm{X}\,\mathrm{D} + 1. \tag{13}$$

in our case. If we want to go across an integral/cointegral operator A, B, we can apply the *First Fundamental Theorem of Calculus* (often formulated in terms of the "indefinite integral"),

$$\begin{aligned} \mathrm{D}\,\mathrm{A} &= 1, \\ \mathrm{D}\,\mathrm{B} &= -1. \end{aligned} \tag{14}$$

The only remaining case for D is now the boundary operators L, R. But this is trivial, because the *derivatives of the boundary constants* must always vanish, so

$$\begin{aligned} \mathrm{D}\,\mathrm{R} &= 0, \\ \mathrm{D}\,\mathrm{L} &= 0. \end{aligned} \tag{15}$$

Now that all the letters D are 'isolated' on the far right, thus giving an iterated differential operator D^k, we come to the candidates for isolation—the boundary operators L, R. Here the idea is similar to that of the differential operator: that it is not economical to extract boundary values except at the very end, namely operating directly on the function given as input to the compound operator. The only difference is that we do not move it completely to the right; it should 'stop' before D. The reason is that it makes perfect sense to have an operator like $L\,D$, meaning "take the derivative of the left boundary point", whereas we have seen above that $D\,L$ simply vanishes. So let us first see what happens when a boundary operator on the left 'hits' an integral/cointegral operator on the right. Obviously, the effect is either to expand the range of integration to the full interval or to collapse it to the empty interval. Hence we have the *integration-transport relations*

$$\begin{aligned} \mathrm{L}\,\mathrm{A} &= 0, \\ \mathrm{R}\,\mathrm{A} &= \mathrm{A} + \mathrm{B}, \\ \mathrm{L}\,\mathrm{B} &= \mathrm{A} + \mathrm{B}, \\ \mathrm{R}\,\mathrm{B} &= 0. \end{aligned} \tag{16}$$

Finally, when they hit an X, they are simply propagated with the corresponding boundary value of $x \mapsto x$ as an additional factor—which is simply 0 or 1 for L and R, respectively. This yields the boundary propagation relations

$$\begin{aligned} \mathrm{L}\,\mathrm{X} &= 0, \\ \mathrm{R}\,\mathrm{X} &= \mathrm{R}. \end{aligned} \tag{17}$$

At this stage, the task of simplification is reduced to putting the remaining A, B and X into a canonical order. We cannot solve this problem completely in the present setup; this will only be done later in this chapter in the frame of a more general noncommutative polynomial ring. But we can still specify the most important interactions, namely how to resolve the clash between two

adjacent integral/cointegral operators. Unlike for differential operators, one can always reduce iterated integrations into a single integral operator with a suitable kernel; see the next section for a more detailed exposition of this idea. The basic tool for doing this is partial integration, yielding the *integral contraction relations*

$$\begin{aligned} \mathrm{A\,A} &= \mathrm{X\,A} - \mathrm{A\,X}, \\ \mathrm{A\,B} &= \mathrm{X\,B} + \mathrm{A\,X}, \\ \mathrm{B\,A} &= \mathrm{A} + \mathrm{B} - \mathrm{X\,A} - \mathrm{B\,X}, \\ \mathrm{B\,B} &= \mathrm{B\,X} - \mathrm{X\,B}. \end{aligned} \tag{18}$$

Assuming that we have finished rearranging the left part of the word, we should finally consider what to do if an integral/cointegral operator 'bumps into' the right part of the word consisting of the boundary and differential operators (an X bumping into them does not harm). Obviously, the integral/cointegral operator and neutralized a differential operator occurring right of it, by virtue of the *Second Fundamental Theorem of Calculus* (often formulated in terms of the "definite integral"),

$$\begin{aligned} \mathrm{A\,D} &= 1 - \mathrm{L}, \\ \mathrm{B\,D} &= \mathrm{R} - 1. \end{aligned} \tag{19}$$

The other cases, dealing with integrations 'bumping into' boundary operators, are rather trivial; they boil down to integrating constant functions. One obtains thus the *boundary integrations*

$$\begin{aligned} \mathrm{A\,L} &= \mathrm{X\,L}, \\ \mathrm{B\,L} &= \mathrm{L} - \mathrm{X\,L}, \\ \mathrm{A\,R} &= \mathrm{X\,R}, \\ \mathrm{B\,R} &= \mathrm{R} - \mathrm{X\,R}. \end{aligned} \tag{20}$$

Combining all these equalities (13), (14), (15), (16), (17), (18), (19), (20), we have now assembled all the relevant *polynomial interaction equalities.*

Joining them with the two left-over polynomial Moore-Penrose equations from (11), we can start to solve for the Green's operator G. Before doing so, however, it is reasonable to ask ourselves whether we can maybe reduce (11) even more, keeping only one of these two equations. In general, they will of course be independent, but now we have got some additional knowledge from the interaction equalities that might warrant a further reduction. As the information about the boundary conditions is encoded in the nullspace projector $P = (1 - X)\,L + X\,R$ occurring only in the third equation, it is clear that the only possible reduction is to infer the fourth equation from the third. Trying this out, we find that the *fourth equation is indeed redundant:* Bringing everything to the left side, the third equation says that

$$\mathrm{G\,D}^2 - 1 + (1 - \mathrm{X})\,\mathrm{L} + \mathrm{X\,R} \tag{21}$$

is equal to the zero polynomial. We multiply (21) by D^2 from the left and by A^2 from the right, then we apply the reduction system induced by the interaction equalities on it. The result is $-1 + G\,D^2$, which must still be equal to the zero polynomial—and this is the fourth Moore-Penrose equation.

Instead of doing this boring reduction by hand, we will *let the computer do the job*. In Chapter 2 we give a detailed description of the computer package used here and developed in the frame of this thesis. For the moment, it is enough to know that we can specify the oriented equalities that should be applied for reduction. We do this in the conventional way used in *Theorema*

[19], namely by stating the equalities together with a label for identifying them in a suitable environment (the label we have chose is simply the monomial on the left-hand side of the equality). Here we have used the environment "System" since we are dealing with a system of equations; for other purposes, one might prefer environments like "Theorem" or "Lemma" (there is no semantic difference between environment names, so their choice is just a matter of style). After initializing the packages, we specify the equality systems to be used and tell the system to apply the reduction engine for noncommutative polynomials as the default evaluator, using the specified interaction equalities. Besides this, we give the list of indeterminates to the evaluator and instruct it to display the computational trace immediately after the Compute call rather than in a separate notebook. All this is summarized in Input 4 below (remember that everything between the caption and the full square at the end is entered verbatim).

4 Input (Computational Setup for Green's Operators)

```
Needs["Theorema`"]
Needs["Theorema`Evaluators`UserEvaluators`GreenEvaluator`"]
System["1. First Equalities for Isolating Differential Operators",
```

$D\,A = 1$	"DA"
$D\,B = -1$	"DB"
$D\,X = 1 + X\,D$	"DX"
$D\,L = 0$	"DL"
$D\,R = 0$	"DR"

]

```
System["2. First Equalities for Isolating Boundary Operators",
```

$L\,A = 0$	"LA"
$R\,A = A + B$	"RA"
$L\,B = A + B$	"LB"
$R\,B = 0$	"RB"
$L\,X = 0$	"LX"
$R\,X = R$	"RX"

]

```
System["3. First Equalities for Contracting Integration Operators",
```

$A\,A = X\,A - A\,X$	"AA"
$A\,B = X\,B + A\,X$	"AB"
$B\,A = A + B - X\,A - B\,X$	"BA"
$B\,B = B\,X - X\,B$	"BB"

]

System[*"4. First Equalities for Absorbing Integration Operators"*,

$A\,D = -L + 1$	"AD"
$B\,D = R - 1$	"BD"
$A\,L = X\,L$	"AL"
$B\,L = L - X\,L$	"BL"
$A\,R = X\,R$	"AR"
$B\,R = R - X\,R$	"BR"

]

```
SetOptions[Compute, by → ReduceNoncommutativePolynomial, using → {
      System["1. First Equalities for Isolating Differential Operators"],
      System["2. First Equalities for Isolating Boundary Operators"],
      System["3. First Equalities for Contracting Integration Operators"],
      System["4. First Equalities for Absorbing Integration Operators"]}];
SetOptions[ReduceNoncommutativePolynomial,
    Indeterminates → {G, D, A, B, X, L, R}, inNotebook → "Current"];
```

□

We start the computation by the generic *Theorema* command *Compute*, applied to the input polynomial (remember that all the output listed below is fully computer-generated). The reduction steps are marked by the label of the equality used.

5 Computation (Derivation of the Fourth Moore-Penrose Equation from the Third)

Compute[$D^2\,(G\,D^2 - 1 + (1 - X)\,L + X\,R)\,A^2$];

We compute:

$$-\mathrm{D}\,\boxed{\mathrm{D\,A}}\,\mathrm{A} + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{D}^2\,\mathrm{X}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{G}\,\mathrm{D}^2\,\mathrm{A}^2 \overset{(DA)}{=}$$

$$-\boxed{\mathrm{D\,A}} + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{D}^2\,\mathrm{X}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{G}\,\mathrm{D}^2\,\mathrm{A}^2 \overset{(DA)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{D}^2\,\mathrm{X}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{G}\,\mathrm{D}\,\boxed{\mathrm{D\,A}}\,\mathrm{A} \overset{(DA)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G}\,\boxed{\mathrm{D\,A}} + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{D}^2\,\mathrm{X}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 \overset{(DA)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G} + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{D}\,\boxed{\mathrm{D\,X}}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 \overset{(DX)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G} - \mathrm{D}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \boxed{\mathrm{D\,X}}\,\mathrm{D}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{X}\,\mathrm{R}\,\mathrm{A}^2 \overset{(DX)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G} - 2\,\mathrm{D}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{X}\,\mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}\,\boxed{\mathrm{D\,X}}\,\mathrm{R}\,\mathrm{A}^2 \overset{(DX)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G} - 2\,\mathrm{D}\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{D}\,\mathrm{R}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{X}\,\mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 + \boxed{\mathrm{D\,X}}\,\mathrm{D}\,\mathrm{R}\,\mathrm{A}^2 \overset{(DX)}{=}$$

$$-1 + \mathrm{D}^2\,\mathrm{G} - 2\,\boxed{\mathrm{D\,L}}\,\mathrm{A}^2 + 2\,\mathrm{D}\,\mathrm{R}\,\mathrm{A}^2 + \mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 - \mathrm{X}\,\mathrm{D}^2\,\mathrm{L}\,\mathrm{A}^2 + \mathrm{X}\,\mathrm{D}^2\,\mathrm{R}\,\mathrm{A}^2 \overset{(DL)}{=}$$

$$-1 + D^2\,G + 2\,D\,R\,A^2 + D\,\boxed{D\,L}\,A^2 - X\,D^2\,L\,A^2 + X\,D^2\,R\,A^2 \overset{(DL)}{=}$$

$$-1 + D^2\,G + 2\,D\,R\,A^2 - X\,D\,\boxed{D\,L}\,A^2 + X\,D^2\,R\,A^2 \overset{(DL)}{=}$$

$$-1 + D^2\,G + 2\,\boxed{D\,R}\,A^2 + X\,D^2\,R\,A^2 \overset{(DR)}{=}$$

$$-1 + D^2\,G + X\,D\,\boxed{D\,R}\,A^2 \overset{(DR)}{=}$$

$$-1 + D^2\,G \qquad \square$$

□

So the four polynomial Moore-Penrose equations really collapse into the single equation

$$G\,D^2 = 1 - (1 - X)\,L - X\,R, \tag{22}$$

and this is the only place where the unknown G enters the equation system. Therefore the task of solving for G boils down to multiplying (22) by a right inverse of D^2 and reducing the corresponding polynomial on the right-hand side in a fashion analogous to that of Computation 5. Observe that we have thus achieved a *significant simplification* of the original task of solving an equation system with four, three or even two equations: In all these cases, one faces the much more difficult question of somehow eliminating the coupling between the various occurrences of G in these equations. In the second Moore-Penrose equation, there is even a non-linear occurrence of G, which would complicate the whole problem considerably: In this case, we would need tools for solving polynomial systems.

But even then we are well within the polynomial approach, and there are powerful (but necessarily more complex) methods available for attacking such problems. The most successful is certainly the computation of *Gröbner bases*, originally developed for *commutative polynomials* by B. Buchberger; see the original PhD thesis [17], the journal version [7], and the concise treatment in [13]. Buchberger's algorithm computes a Gröbner basis for the ideal of any given set of polynomials; using a lexicographic term ordering, the Gröbner basis allows to read off the solutions of the corresponding polynomial system.

Buchberger's algorithm has a direct analog in the *noncommutative setting*, known as Mora's algorithm [51]. Unlike the commutative case, it does not always terminate. But when it does, the Gröbner basis thus computed enjoys similar properties as the commutative ones. Hence it would be applicable to our present problem. We followed this strategy in the paper [57] because at that time we had not yet seen the possibility of reducing everything to one equation. So it seems that we do not need Gröbner bases, but the real fact is that we can avoid the costly completion algorithm (Buchberger's / Mora's algorithm) while there is still a Gröbner basis behind the scenes. The point is just that it does not change; it need not be "completed". See the remarks after Definition 30 for more details.

Now finding a right inverse is next to trivial in our case: Equation (14) tells us that D has the right inverse A, so D^2 has the right inverse A^2. Multiplying (22) by this right inverse, we obtain—more or less—the desired Green's operator

$$G = (1 - (1 - X)\,L - X\,R)\,A^2, \tag{23}$$

just that it is 'in a slightly unusual form'. In particular, this is not yet a (single) integral operator with a Green's function as its kernel. Let us therefore apply the reduction system induced by the interaction equalities for trying to get a better representation for G. Again, we prefer to resort to the automatic evaluator implemented in the frame of this thesis.

6 Computation (Reduction of the Green's Operator)

```
Compute[(1 - (1 - X) L - X R) A^2];
```

We compute:

$$A^2 - \boxed{L\,A}\,A + X\,L\,A^2 - X\,R\,A^2 \overset{(LA)}{=}$$

$$A^2 + X\,\boxed{L\,A}\,A - X\,R\,A^2 \overset{(LA)}{=}$$

$$A^2 - X\,\boxed{R\,A}\,A \overset{(RA)}{=}$$

$$\boxed{A^2} - X\,A^2 - X\,B\,A \overset{(AA)}{=}$$

$$-A\,X + X\,A - X\,\boxed{A^2} - X\,B\,A \overset{(AA)}{=}$$

$$-A\,X + X\,A - X^2\,A + X\,A\,X - X\,\boxed{B\,A} \overset{(BA)}{=}$$

$$-A\,X - X\,B + X\,A\,X + X\,B\,X \qquad \square$$

□

The computation yields the Green's operator in the canonical form

$$G = -A\,X - X\,B + X\,A\,X + X\,B\,X, \tag{24}$$

and we could already regard this as the final answer. If we prefer to see the *Green's function*, we collect in (24) the terms belonging to the same integration operator, giving

$$G = (X - 1)\,A\,X + X\,B\,(X - 1). \tag{25}$$

Now we simply unfold the definition of the operators A, B, X as they are applied to a function $f \in C[0, 1]$, evaluated at a point $x \in [0, 1]$, so

$$\begin{aligned} Gf(x) &= (x - 1)\int_0^x \xi\, f(\xi)\, d\xi + x\int_x^1 (\xi - 1)\, f(\xi)\, d\xi = \\ &\int_0^x (x - 1)\,\xi\, f(\xi)\, d\xi + \int_x^1 x\,(\xi - 1)\, f(\xi)\, d\xi. \end{aligned} \tag{26}$$

Writing this as a single integral such that (4) holds, we must pack both integrands into one function, which we define by the corresponding case distinction

$$g(x, \xi) = \begin{cases} (x-1)\,\xi & \Leftarrow \quad 0 \le \xi \le x \le 1, \\ x\,(\xi - 1) & \Leftarrow \quad 0 \le x \le \xi \le 1. \end{cases} \tag{27}$$

It should be noted that the procedure of transforming G in (24) into the corresponding Green's function g in (27) is completely *mechanical:* The monomials with A go into the first branch of g, those with B into the second branch; the occurrences of X before the integration operators become x, those after the integration operators become ξ. In fact, such a "mechanical procedure" could readily be implemented; see the discussion towards the end of Section 2 of Chapter 2.

1.2 The Green's Polynomials

The goal of this section is to generalize the solution strategy used for the example of steady heat conduction in a rod. We want to treat any regular BVPs for linear differential operators with constant coefficients. Let us make this precise by the following definition.

7 Definition (Regular BVP)

Let $[a, b]$ be a finite interval in $\mathbb{R}$, and let T be an n-th order linear differential operator with constant coefficients

$$T\,u = c_0\,u + c_1\,u' + c_2\,u'' + \ldots + c_{n-1}\,u^{(n-1)} + u^{(n)}$$

operating on the Banach space $(C^\infty[a, b], \|\cdot\|_\infty)$. Let $B_1, \ldots, B_n$ be boundary operators defined on the same domain, say

$$B_i\,u = p_{i,0}\,u^{(n)}(a) + \ldots + p_{i,n-1}\,u'(a) + p_{i,n}\,u(a) + q_{i,0}\,u^{(n)}(b) + \ldots + q_{i,n-1}\,u'(b) + q_{i,n}\,u(b),$$

with coefficients $p_{i,0}, \ldots, p_{i,n}, q_{i,0}, \ldots, q_{i,n} \in \mathbb{R}$ for each $i = 1, \ldots, n$. The *boundary value problem* induced by T and $B_1, \ldots, B_n$ is to find for each forcing function $f \in C^\infty[a, b]$ a function $u \in C^\infty[a, b]$ such that

$$T\,u = f,$$
$$B_1\,u = \ldots = B_n\,u = 0.$$

We call the BVP *regular* iff it has a unique solution $u \in C^\infty[a, b]$ for each forcing function $f \in C^\infty[a, b]$.

□

This BVP is actually inhomogeneous in the differential equation and homogeneous in the boundary conditions (a so-called *semi-inhomogeneous* problem). But we can always decompose a fully inhomogeneous problem into such a semi-inhomogeneous one and a rather trivial BVP with homogeneous differential equation and inhomogeneous boundary conditions (a so-called *semi-homogeneous* problem); see page 43 in [63]. Hence we have not lost any essential generality in Definition 7 by assuming homogeneous boundary conditions.

Applying the solution strategy used in the example of steady heat conduction, we must first compute a suitable projector P onto $\mathcal{N}(T)$ such that the boundary conditions will be fulfilled for $T_{P,I}^{\dagger}$. From the theory of ordinary differential equations we know that one can obtain a fundamental system for a linear differential operator with constant coefficients by taking the functions $x \mapsto e^{\lambda x}$, $x \mapsto x\,e^{\lambda x}$, …, $x \mapsto x^{k-1}\,e^{\lambda x}$ for each k-fold root $\lambda \in \mathbb{C}$ of the characteristic equation for T; see

page 89 in [21]. They form the basis of a function algebra that will be dubbed the *polyexponential functions* because they are a mixture of "ordinary polynomial functions" (having the functions $x \mapsto x^k$ as a basis) and "exponential polynomials" (having the functions $x \mapsto e^{\lambda x}$ as a basis). For the sake of brevity, we will usually refer to them as the "polyexponentials", and we will denote them by $\mathcal{E}$.

Then we must introduce *new indeterminates* for the multiplication operators induced by the powers $x \mapsto x^k$ (where k runs through $\mathbb{N}^*$) and by the exponentials $x \mapsto e^{\lambda x}$ (where λ runs through $\mathbb{C}^*$). The resulting noncommutative polynomial ring will have infinitely many indeterminates, but this is no problem since each individual polynomial can contain only finitely many of them. So we will work in the structure

$$\mathbb{C}\langle\{D, A, B, L, R\} \bigcup \{X^k \mid k \in \mathbb{N}^*\} \bigcup \{E_\lambda \mid \lambda \in \mathbb{C}^*\}\rangle, \tag{28}$$

where X^k denotes the corresponding multiplication operator mapping each $u \in C^\infty[a, b]$ to

$$X^k\, u = x \mapsto x^k\, u\,(x), \tag{29}$$

and likewise E_λ denotes the operator mapping $u \in C^\infty[a, b]$ to

$$E_\lambda\, u = x \mapsto e^{\lambda x}\, u\,(x). \tag{30}$$

Of course we must now add some interaction relations describing that powers and exponentials may be contracted and that we may always order them such that the exponentials come first. We call them *algebraic interaction equalities* as they do not refer to any analysis concepts like differentiation or integration. Besides these new interactions, we have to make some obvious modifications in the isolation equalities (still assuming $a = 0$ and $b = 1$ to simplify things for the moment).

Before specifying the interaction equalities, we tell the system that we want to use the *powers* of X as indeterminates: For example, when X^3 appears in a polynomial, this is *one* atomic indeterminate; whereas D^3 is understood as a shorthand for $D\,D\,D$.

Then we list all the interactions we need, plus some *built-in knowledge*: The addition and subtraction appearing in the parameters of X^k and E_λ are not the same as the plus and minus connecting the polynomials! The former should be left untouched by Mathematica, whereas the latter are to be executed in the expected manner. Thus we want to have $X^{2\oplus 3}$ simplify to X^5. Therefore we use circled operator symbols for those operations that should be carried out by Mathematica, and we tell the system explicitly that $\oplus$ and $\ominus$ should be the Mathematica functions Plus and Minus, respectively. This is the meaning of the environment **Built-in**[*"Arithmetic"*].

Having specified all the explicit and implicit knowledge, we must *fix some additional technicalities*. First we instruct the Compute command to use the new set of interaction equalities, then we tell the evaluator the names of the current indeterminates and some abbreviations (see Chapter 2 for details). The option ReductionPhases of the function ReduceNoncommutativePolynomial specifies the order in which the systems given below are to be applied; the algebraic equalities are not mentioned, meaning that they are to be used throughout all the phases. As we do not want to see the long and tedious trace, we set the corresponding option to "None" this time.

8 Input (Modified Interaction Equalities for Exponential Polynomials)

UsePowers[X]

System[*"First Equalities for Algebraic Simplication"*, any[i, j, λ, μ],

$X^i\, X^j = X^{i \oplus j}$	"XX"
$E_\lambda\, E_\mu = E_{\lambda \oplus \mu}$	"EE"
$X^i\, E_\lambda = E_\lambda\, X^i$	"XE"

]

System[*"1. First Equalities for Isolating Differential Operators"*, any[i, λ],

$D\, A = 1$	"DA"
$D\, B = -1$	"DB"
$D\, X^i = i\, X^{i \ominus 1} + X^i\, D$	"DX"
$D\, E_\lambda = \lambda\, E_\lambda + E_\lambda\, D$	"DE"
$D\, L = 0$	"DL"
$D\, R = 0$	"DR"

]

System[*"2. First Equalities for Isolating Boundary Operators",* any[i, λ],

$L\, A = 0$	"LA"
$R\, A = A + B$	"RA"
$L\, B = A + B$	"LB"
$R\, B = 0$	"RB"
$L\, X^i = 0$	"LX"
$R\, X^i = R$	"RX"
$L\, E_\lambda = L$	"RE"
$R\, E_\lambda = E\, R$	"RE"

]

System[*"3. First Equalities for Contracting Integration Operators"*,

$A\, A = X\, A - A\, X$	"AA"
$A\, B = X\, B + A\, X$	"AB"
$B\, A = A + B - X\, A - B\, X$	"BA"
$B\, B = B\, X - X\, B$	"BB"

]

System[*"4. First Equalities for Absorbing Integration Operators"*,

$A\, D = -L + 1$	"AD"
$B\, D = R - 1$	"BD"
$A\, L = X\, L$	"AL"
$B\, L = L - X\, L$	"BL"
$A\, R = X\, R$	"AR"
$B\, R = R - X\, R$	"BR"

]

```
Built-in["Arithmetic",
  ⊕ → Plus
  ⊖ → Minus]
SetOptions[Compute,
  by → ReduceNoncommutativePolynomial,
  using → {
      System["First Equalities for Algebraic Simplication"],
      System["1. First Equalities for Isolating Differential Operators"],
      System["2. First Equalities for Isolating Boundary Operators"],
      System["3. First Equalities for Contracting Integration Operators"],
      System["4. First Equalities for Absorbing Integration Operators"]},
  built-in → Built-in["Arithmetic"]
];
SetOptions[ReduceNoncommutativePolynomial,
  ReductionPhases → {
      "1. First Equalities for Isolating Differential Operators",
      "2. First Equalities for Isolating Boundary Operators",
      "3. First Equalities for Contracting Integration Operators",
      "4. First Equalities for Absorbing Integration Operators"},
  Indeterminates → {X□, D, L, R, E□, A, B}, Units → {X^0, E0}, inNotebook → "None"];
```

□

Using the reduction system resulting from orienting these equalities from left to right, we obtain a fairly universal method for simplifying a polynomial in (28). For example, we can easily compute the fifth Legendre polynomial, whose Rodriguez formula is given by

$$\mathrm{P}_3\,(\mathrm{x}) = \frac{1}{2^3\,3!}\,\partial_{\mathrm{x}}^3\,(\mathrm{x}^2-1)^3$$

The analytic polynomial corresponding to (30) is

$$\mathrm{D}^3\,(\mathrm{X}^2-1)^3 \tag{31}$$

times the normalization factor, and we can simplify (31) in the usual way for getting its expanded version. (We have left out the normalization factor only because it clutters the output.)

9 Computation (Sample Reduction of an Analytic Polynomial)

```
Compute[D^3 (X^2 - 1)^3]
```

$$-72\,X + 18\,D + 120\,X^3 - 108\,X^2\,D + 90\,X^4\,D - 36\,X^3\,D^2 - D^3 + 18\,X\,D^2 + 18\,X^5\,D^2 + 3\,X^2\,D^3 - 3\,X^4\,D^3 + X^6\,D^3$$

□

The polynomial (31) is of course much more complex than the ordinary Legendre polynomial since it represents a differential operator obtained by operating ∂_x not only on the coefficient function $(x^2-1)^3$ but also 'across' it. However, we get back the original Legendre polynomial—in expanded form—by formally substituting $D \leftarrow 0$ and dividing by 48, thus obtaining

$$P_3\,(x) = \frac{1}{2}\,(5\,x^3 - 3\,x).$$

Having the new polynomial ring (28), we should try to close the gaps mentioned after (17). The overall strategy explained there still works in the new polynomial ring: We carry out the reduction in four phases—first isolating the differential operators, second isolating the boundary operators,

third contracting the integration operators, and fourth absorbing them into each other (the algebraic interactions may be used in any phase). But now we face a *problem that may occur in the third phase*: How can we contract two integration operators, for example an A and another A if there are some algebraic operators E_λ and X^k in between? Merging these intermediate operators yields a single multiplication operator M_f induced by some basis polyexponential f. Let us see how we can absorb the two integral operators into one by using partial integration. For any $u \in C^\infty[a, b]$, we have

$$(\mathrm{A}\,\mathrm{M_f}\,\mathrm{A})\,\mathrm{u} = \mathrm{x} \mapsto \int_\mathrm{a}^\mathrm{x} \mathrm{f}(\xi)\,\mathrm{Au}\,(\xi)\,d\xi \tag{32}$$

according to the standard interpretation of the indeterminates A and M_f as operators on the Banach space $C^\infty[a, b]$. Applying partial integration to this integral, we have to choose *some* antiderivative of f. In particular, we may choose $A\,f$, the antiderivative normed to be 0 at a. Note, however, that $A\,f$ is simply another polyexponential, not to be confused with the noncommutative polynomial $A\,M_f$. Let us therefore write $\int^* f$ for this operation of A on f and let us call it the *integral action*. Likewise we will write $\int_* f$ for the corresponding operation of B on f and call it the *cointegral action*. We will clarify these issues in greater detail below in Definition 11. Using the antiderivative $\int^* f$ of f, partial integration yields

$$\int_0^\mathrm{x} \mathrm{f}(\xi)\,\mathrm{Au}\,(\xi)\,d\xi = \left(\int^* \mathrm{f}\right)(\xi)\,\mathrm{Au}\,(\xi)\,\Big]_{\xi=\mathrm{a}}^\mathrm{x} - \int_\mathrm{a}^\mathrm{x} \left(\int^* \mathrm{f}\right)(\xi)\,\mathrm{u}\,(\xi)\,d\xi, \tag{33}$$

so the operator in (32) is

$$\mathrm{A}\,\mathrm{M_f}\,\mathrm{A} = \mathrm{M}_{\int^* \mathrm{f}}\,\mathrm{A} - \mathrm{A}\,\mathrm{M}_{\int^* \mathrm{f}}, \tag{34}$$

where f and accordingly $\int^* f$ are some concrete polyexponentials. From (34) we can extract polynomial equalities by substituting the three possible cases for M_f, namely X^k for pure polynomials and E_λ for pure exponentials and $E_\lambda\,X^k$ for proper polyexponentials. So the only problem is to evaluate the integrals of $x \mapsto x^k$ and $x \mapsto e^{\lambda x}$ and $x \mapsto e^{\lambda x}\,x^k$, respectively. Now the first two are trivial, but looking up integration tables for the third reveals some recursion formulae. Therefore we conclude that we can always carry out the reduction (34) from left to right for any concrete exponential polynomial like $A\,E_3\,X^4\,A$, but we cannot write down a closed formula for the generic instance $E_\lambda\,X^k$.

How can we solve this problem? Well, why not take (34) itself as the new interaction equality! In other words, we can generalize the polynomial ring once more such that it includes *all multiplication operators* M_f induced by functions f out of a certain reasonable class $\mathcal{F}$ that can be adapted; the default choice for $\mathcal{F}$ will be the polyexponentials. Of course, then we do not need the indeterminate X^k and E_λ anymore; they are subsumed by $M_{x \mapsto x^k}$ and $M_{x \mapsto e^{\lambda x}}$, respectively. Since it starts to get tedious by now, let us introduce a convenient agreement for getting rid of all these cumbersome lambda quantifiers. So from now on, we will speak of the *functions* x^k and $e^{\lambda x}$, and so their corresponding indeterminates are M_{x^k} and $M_{e^{\lambda x}}$.

10 Convention (Implicit Lambda Quantification)

If a term T appears where we should have a function, it should be understood as $x \mapsto T$.

□

The polyexponentials $\mathcal{E}$ constitute just one out of various conceivable 'function domains' that can be used for multiplication operators, but in the frame of this thesis we will not consider other possibilities. Going through the interaction equalities of Input 8, we see that the only essential thing is that we have some set of objects—typically but not always functions—with 'reasonably behaving' operations for sums, products, derivatives, integrals and boundary values. For the first two of them, there seems to be a natural choice for specifying what we mean by "reasonable", namely the *ring axioms*.

However, we need slightly more. Obviously, the multiplication operators M_{f+g} and $M_f + M_g$ do the same under any 'reasonable' interpretation, so we would have to add the interaction equalities $M_{f+g} = M_f + M_g$, basically a restatement of distributivity. We can also interpret this interaction as saying that we do not really need the indeterminates M_{f+g} as soon as we have M_f and M_g. Generally speaking, it is sufficient to consider only those M_f where f is irreducible with respect to addition; in other words, f should range over a basis of the function domain, which thus turns out to be an *vector space*—just as its prototype $\mathcal{E}$. In fact, it must even be an algebra like $\mathcal{E}$ is because we also have an appropriate multiplicative structure on it.

Just as for the algebraic operations (addition, multiplication), there are also natural axioms for the *analytic operations:* differentation $f \mapsto f'$, the integral action $f \mapsto \int^* f$, the cointegral action $f \mapsto \int_* f$, the left boundary action $f \mapsto f^{\leftarrow}$, and the right boundary action $f \mapsto f^{\rightarrow}$. For example, the essential properties of differentiation alone are the linearity rule and the product rule; adding them to a ring / algebra gives what is usually called a differential ring / algebra. We have chosen the name "analytic algebra" because we extend the notion of algebra not only by differentiation but also by the second fundamental concept of analysis, namely that of integration.

The axioms for integrations and boundary values are formulated in the same spirit as the linearity rule and the product rule for differential algebras. In fact, we obtained all of the axioms specified below very naturally by attempting the proof of Theorem 28 without them and adding what is needed until the proof succeeds (in the *Theorema* project, we are working on this approach in the larger context of theorem proving, following what B. Buchberger calls the *lazy thinking paradigm*). The result of this process is contained in the definition given below. Note that we have used Convention 10 for *suppressing lambda quantification* in these axioms. For example, the term $f - f^{\leftarrow}$ is to be understood as the function $x \mapsto f(x) - f^{\leftarrow}$.

11 Definition (Analytic Algebra)

An algebra $\mathcal{F}$ is called an *analytic algebra* iff it has unary linear operations differentiation $' : \mathcal{F} \to \mathcal{F}$, integral $\int^* : \mathcal{F} \to \mathcal{F}$, cointegral $\int_* : \mathcal{F} \to \mathcal{F}$, left boundary value $^{\leftarrow} : \mathcal{F} \to \mathbb{C}$ and right boundary value $^{\rightarrow} : \mathcal{F} \to \mathbb{C}$, subject to the postulates listed below.

Axioms["*Analytic Algebra*", any[f, g],

$(f\,g)' = f'\,g + f\,g'$	"dm"
$\int^* f' = f - f^{\leftarrow}$	"ad"
$\int_* f' = f^{\rightarrow} - f$	"bd"
$\left(\int^* f\right)' = f$	"da"]
$\left(\int_* f\right)' = -f$	"db"
$(f\,g)^{\leftarrow} = f^{\leftarrow}\,g^{\leftarrow}$	"lm"
$(f\,g)^{\rightarrow} = f^{\rightarrow}\,g^{\rightarrow}$	"rm"

□

For understanding the *axioms of an analytic algebra*, we observe the following. An analytic algebra is essentially a differential algebra with two generalized inverses for the derivation '. Comparing axioms "ad", "da" with the Moore-Penrose equations in Proposition 3, we see that the integral is indeed a generalized inverse for ', with trivial range projector (since $C^\infty[0, 1]$ remains all of $C^\infty[0, 1]$ under differentiation) and the left boundary action as nullspace projector (note the lambda convention). For the cointegral, things are similar; the only difference is that now we have a generalized *skew* inverse in the sense that the *negative* cointegral is a generalized inverse for ', with trivial range projector and right boundary action. So the operations $^\leftarrow$ and $^\rightarrow$ just serve to choose among the generalized inverses by fixing the integration constant. Axioms "lm" and "rm" stipulate that $f \mapsto (x \mapsto f^\leftarrow)$ and $f \mapsto (x \mapsto f^\rightarrow)$ be homomorphisms in the algebra $\mathcal{F}$.

Having the concept of analytic algebra, we can finally introduce the *analytic polynomials* as announced before. Note that the multiplication operators are only indexed over a *basis* of the given analytic algebra, thus guaranteeing an economic representation as explained above.

12 Definition (Analytic Polynomials)

Let $\mathcal{F}$ be an analytic algebra. Then

$$\mathbb{C}\langle\{D, A, B, L, R\} \bigcup \{M_f \mid f \in \mathcal{F}^\#\}\rangle$$

will be called the ring of *analytic polynomials* over $\mathcal{F}$, denoted by $\mathcal{An}(\mathcal{F})$. If $\mathcal{F}$ is not mentioned, it is assumed to be the system $\mathcal{E}$ of polyexponentials; the corresponding polynomial ring is then denoted by $\mathcal{An}$. Similar conventions will be assumed for other occurrences of $\mathcal{F}$ and $\mathcal{E}$ in the subsequent text.

□

At this point, we should also mention an alternative way of introducing analytic polynomials. Looking at equality (34), we could also view the multiplication operators as coefficients instead of indeterminates; then we would have $\mathrm{A\,f\,A} = (\int^* \mathrm{f})\,\mathrm{A} - \mathrm{A} \int^* \mathrm{f}$. Of course, these coefficients do not commute with the indeterminates, so here we are dealing with the *uncommutative polynomials* of $\mathcal{P}_{\mathcal{UniRing}}(\mathcal{F}, \{D, A, B, L, R\})$; see Section 5 of the Appendix. But since the coefficients do commute with $\mathbb{R}$, we would have to factor out these commutations. Such a formulation by uncommutative polynomials has the advantage that one does not need infinitely many indeterminates $\{\mathrm{M_f} \mid \mathrm{f} \in \mathcal{F}_\#\}$ in the polynomial ring. Nevertheless, we will stick to the formulation of Definition 12 because the infinitude of $\{\mathrm{M_f} \mid \mathrm{f} \in \mathcal{F}_\#\}$ does not cause serious problems whereas uncommutative polynomials are not well studied in the literature.

Equality (34) can now be understood as a polynomial interaction equality, as was desired. Strictly speaking, though, this formulation is still not correct. The problem is that $\int^* f$ might not be in $\mathcal{F}^\#$ anymore even though f was; in fact, this is usually so in the case in the prototype algebra $\mathcal{T}$. But since $\mathcal{F}$ is an analytic algebra, $\int^*$ is an operation within $\mathcal{F}$, so we can write $\int^* f$ as a linear combination $\lambda_1 f_1 + \ldots + \lambda_n f_n$ with $f_1, \ldots, f_n \in \mathcal{F}^\#$ and $\lambda_1, \ldots, \lambda_n \in \mathbb{C}$. Hence we can write $\lambda_1 M_{f_1} + \ldots + \lambda_n M_{f_n}$ instead of the wrong $M_{\int^* f}$. This operation can be extended to all of $\mathcal{An}(\mathcal{F})$ by applying it to all the multiplication indeterminates of a given polynomial and then expanding the result. In particular, we have $M_\lambda = \lambda$ for multiplication operators induced by constant functions $x \mapsto \lambda$ with $\lambda \in \mathbb{C}$. We will refer to this process as the *basis expansion* in $\mathcal{F}$.

In the actual computations, we do this together with the usual expansion rules for polynomials like distributivity, namely each time after applying a polynomial equation. In the Compute call, this is specified by using the special built-in referred to as *$BasisExpansion*. The other built-in

operations like carrying out integrals and cointegrals are collected in **Built-in**[*"Action operators"*]. So if one wants to use both of these built-in simplifications, one has to specify the option **built-in→** {*$BasisExpansion*, **Built-in**[*"Action operators"*]} of the **Compute** function. See Input 14 below for an example and Chapter 2 for more details about the computation engine.

Since the multiplication operators occur quite often in a typical computation, we will adopt a smoother notation for them. In fact, it starts to become boring to see all these M symbols when the really important information is contained in the indices. Therefore we will lift them to the "ceiling", using the so-called ceiling brackets.

13 Convention (Ceiling Notation)

The notation $\lceil f \rceil$ is a shorthand for the indeterminate M_f of $\mathcal{An}(\mathcal{F})$.

□

Within the noncommutative polynomial ring $\mathcal{An}(\mathcal{F})$, we can now formulate *all* the interactions that will be necessary to reduce an arbitrary polynomial to a simplified form that is even canonical as we will prove in Theorem 28. But first let us review the four phases on an informal basis. In the *first phase*, we isolate all differential operators on the far right. Basically, all the interaction stay the same, but the product rule is now formulated in full generality as

$$\mathrm{D}\lceil \mathrm{f} \rceil = \lceil \mathrm{f} \rceil \mathrm{D} + \lceil \mathrm{f}' \rceil, \tag{35}$$

where $\lceil f' \rceil$ splits again into a linear combination of multiplication operators over $\mathcal{F}$. Intuitively, it is clear that we can *always* perform this step successfully because there are no other 'hurdles' to pass by except A, B, f, L, R.

The *second phase* is also successful in this sense, isolating all boundary operators at the position next to the differential operators (if there is no differential operator, this is the far right). Let us see what happens when we move to the right. Now the differential operators are already out of the way. If we 'hit' an integration operator, we can still apply the integration-transport relations (16). Moving across a multiplication operator $\lceil f \rceil$ is analogous to moving across X^k or E_λ; we simply evaluate the function f at the left or right boundary point, expressed by $f^{\leftarrow}$ and $f^{\rightarrow}$, respectively. Now the only remaining possibility is that one boundary operator meets another, which is so trivial that we have even left it out up to now. For the sake of completeness, though, we will also add it now as the boundary idempotence relations

$$\begin{aligned} \mathrm{L}\,\mathrm{L} &= \mathrm{L}, \\ \mathrm{L}\,\mathrm{R} &= \mathrm{R}, \\ \mathrm{R}\,\mathrm{L} &= \mathrm{L}, \\ \mathrm{R}\,\mathrm{R} &= \mathrm{R}. \end{aligned} \tag{36}$$

Now we come to the *third phase*, contracting integral operators. This is where we had to leave some gaps in the previous reduction systems. So let us analyze this situation carefully. All the differential and boundary operators are already moved to the far right, so we need only be concerned about the interactions between A, B and f. As the cointegral operator is dual and hence analogous to integral operator, the essential question is how to interrelate integration and multiplication operators. As mentioned before, it is always possible to contract all integrations into a single one—unlike differentiations. We have seen an example in (34), contracting to integral operators with a multiplication operator in between. The other three cases with $A\lceil f \rceil B$, $B\lceil f \rceil A$, $B\lceil f \rceil B$ are indeed analogous, as one can see below. The collision of two such integration operators without a

multiplication operator in between, already covered in (18), is subsumed by choosing $f = 1$. However, it is still useful to include them as 'optimized rules' for frequently occurring special cases. Putting these things together, we should now have the intuition that the third phase must also be successful: Whenever the prefix left of the differential/boundary part contains more than one integration operator, we use the interactions just described for contracting them. Finally, we are left with something of the form $\lceil f \rceil A \lceil g \rceil$ or $\lceil f \rceil B \lceil g \rceil$, which can be understood as a suitable integral operator with the "separated kernel" being $f(x)\, g(\xi)$ as in (26).

The *fourth phase* takes care of integral operators being 'cancelled' by differential operators or 'swallowed' by boundary operators. For this purpose, we can again take over the corresponding interactions (19) and (20). Besides this, there are analogous interactions for the monomials $A \lceil f \rceil L$, $B \lceil f \rceil L$, $A \lceil f \rceil R$, $B \lceil f \rceil R$. But we have not yet considered the possibilities $A \lceil f \rceil D$ and $B \lceil f \rceil D$. Let us consider the first case in detail. According to the standard interpretation of A, we have

$$\mathrm{A}\,\lceil \mathrm{f} \rceil\,\mathrm{D}\,\mathrm{u} = \mathrm{x} \mapsto \int_{\mathrm{a}}^{\mathrm{x}} \mathrm{f}(\xi)\,\mathrm{u}'(\xi)\,d\xi \tag{37}$$

for all $u \in C^{\infty}[0, 1]$. (We are now assuming general boundary points a and b.) Applying again partial integration, we can rewrite the integral on the right as

$$\begin{aligned} \int_{\mathrm{a}}^{\mathrm{x}} \mathrm{f}(\xi)\,\mathrm{u}'(\xi)\,d\xi = {} & \mathrm{f}(\xi)\,\mathrm{u}(\xi)\;]_{\xi=\mathrm{a}}^{\mathrm{x}} - \int_{\mathrm{a}}^{\mathrm{x}} \mathrm{f}'(\xi)\,\mathrm{u}(\xi)\,d\xi = \\ & \mathrm{f}(\mathrm{x})\,\mathrm{u}(\mathrm{x}) - \mathrm{f}(\mathrm{a})\,\mathrm{u}(\mathrm{a}) - \int_{\mathrm{a}}^{\mathrm{x}} \mathrm{f}'(\xi)\,\mathrm{u}(\xi)\,d\xi, \end{aligned} \tag{38}$$

so we have

$$\mathrm{A}\,\lceil \mathrm{f} \rceil\,\mathrm{D} = -\mathrm{f}^{\leftarrow}\,\mathrm{L} + \lceil \mathrm{f} \rceil - \mathrm{A}\,\lceil \mathrm{f}' \rceil. \tag{39}$$

Of course, one can derive an analogous interaction equality for $B \lceil f \rceil D$. It is now clear that the fourth phase is also successful in the following sense: It makes sure that we cannot have a monomial containing both a differential and an integration operator. With other words, the operators represented by such monomials are either differential operators (containing only D) or integral operators (containing either only A or only B) or algebraic operators (containing neither D nor A nor B). We will soon make these ideas more precise by proving that the new interaction equalities lead to normal forms in the sense sketched above.

As before there are also *algebraic interactions* (again to be applied throughout all phases), taking care of reduction within $\mathcal{F}$. But now we can formulate them in a single equation that subsumes all of the previous interactions for $X^i\, X^j$, $E_\lambda\, E_\mu$ and $X^i\, E_\lambda$.

So let us *summarize the new collection of interaction equalities* in suitable *Theorema* environments. As we do not use the parametrized indeterminate X^k anymore, we revoke the power convention for X explicitly (just to make sure). In all the equations below, the universal quantifiers for f and g implicitly range over all functions in $\mathcal{F}^{\#}$. After specifying the polynomial equalities, we list the built-in operations for executing the action operators. Then we set some defaults for the functions Compute and ReduceNoncommutativePolynomial, in a manner analogous to what we had before.

14 Input (Interaction Equalities for Analytic Polynomials)

DoNotUsePowers[X]

System[*"Equalities for Algebraic Simplication"*, any[f, g],

$\lceil f \rceil \lceil g \rceil = \lceil f\, g \rceil$	"MM"

]

System[*"1. Equalities for Isolating Differential Operators"*, any[f],

$D\, A = 1$	"DA"
$D\, B = -1$	"DB"
$D \lceil f \rceil = \lceil f \rceil D + \lceil f' \rceil$	"DM"
$D\, L = 0$	"DL"
$D\, R = 0$	"DR"

]

System[*"2. Equalities for Isolating Boundary Operators"*, any[f],

$L\, A = 0$	"LA"
$R\, A = A + B$	"RA"
$L\, B = A + B$	"LB"
$R\, B = 0$	"RB"
$L \lceil f \rceil = f^{\leftarrow}\, L$	"LM"
$R \lceil f \rceil = f^{\rightarrow}\, R$	"RM"
$L\, L = L$	"LL"
$L\, R = R$	"LR"
$R\, L = L$	"RL"
$R\, R = R$	"RR"

]

System[*"3. Equalities for Contracting Integration Operators"*, any[f],

$A \lceil f \rceil A = \left\lceil \int^{*} f \right\rceil A - A \left\lceil \int^{*} f \right\rceil$	"AMA"
$A \lceil f \rceil B = \left\lceil \int^{*} f \right\rceil B + A \left\lceil \int^{*} f \right\rceil$	"AMB"
$B \lceil f \rceil A = \left\lceil \int_{*} f \right\rceil A + B \left\lceil \int_{*} f \right\rceil$	"BMA"
$B \lceil f \rceil B = \left\lceil \int_{*} f \right\rceil B - B \left\lceil \int_{*} f \right\rceil$	"BMB"
$A\, A = \left\lceil \int^{*} 1 \right\rceil A - A \left\lceil \int^{*} 1 \right\rceil$	"AA"
$A\, B = \left\lceil \int^{*} 1 \right\rceil B + A \left\lceil \int^{*} 1 \right\rceil$	"AB"
$B\, A = \left\lceil \int_{*} 1 \right\rceil A + B \left\lceil \int_{*} 1 \right\rceil$	"BA"
$B\, B = \left\lceil \int_{*} 1 \right\rceil B - B \left\lceil \int_{*} 1 \right\rceil$	"BB"

]

System[*"4. Equalities for Absorbing Integration Operators"*, any[f],

$A\lceil f\rceil D = -f^{\leftarrow} L + \lceil f\rceil - A\lceil f'\rceil$	"AMD"
$B\lceil f\rceil D = f^{\rightarrow} R - \lceil f\rceil - B\lceil f'\rceil$	"BMD"
$A D = -L + 1$	"AD"
$B D = R - 1$	"BD"
$A\lceil f\rceil L = \left\lceil \int^{*} f \right\rceil L$	"AML"
$B\lceil f\rceil L = \left\lceil \int_{*} f \right\rceil L$	"BML"
$A\lceil f\rceil R = \left\lceil \int^{*} f \right\rceil R$	"AMR"
$B\lceil f\rceil R = \left\lceil \int_{*} f \right\rceil R$	"BMR"
$A L = \left\lceil \int^{*} 1 \right\rceil L$	"AL"
$B L = \left\lceil \int_{*} 1 \right\rceil L$	"BL"
$A R = \left\lceil \int^{*} 1 \right\rceil R$	"AR"
$B R = \left\lceil \int_{*} 1 \right\rceil R$	"BR"

]

Built-in[*"Action Operators"*,

```
  ™LeftBoundaryValue → LeftBoundaryValue
  ™RightBoundaryValue → RightBoundaryValue
  ™Derivative1 → Derivative1                    ]
  ™IndefiniteIntegral → IndefiniteIntegral
  ™IndefiniteCointegral → IndefiniteCointegral
SetOptions[Compute,
  by → ReduceNoncommutativePolynomial,
  using → {
      System["Equalities for Algebraic Simplication"],
      System["1. Equalities for Isolating Differential Operators"],
      System["2. Equalities for Isolating Boundary Operators"],
      System["3. Equalities for Contracting Integration Operators"],
      System["4. Equalities for Absorbing Integration Operators"]},
  built-in → {$BasisBuiltin, Built-in["Action Operators"]}
];
SetOptions[ReduceNoncommutativePolynomial,
  ReductionPhases → {
      "1. Equalities for Isolating Differential Operators",
      "2. Equalities for Isolating Boundary Operators",
      "3. Equalities for Contracting Integration Operators",
      "4. Equalities for Absorbing Integration Operators"},
  Indeterminates → {D, L, R, A, B, ⌈□⌉}, inNotebook → "None"];
```

□

We will also refer to the above collection as the system of *Green's equalities.*

As a first *example*, let us see how Computation 6 looks like in the new ring $\mathcal{An}$. Then let us try the problematic case that forced us to introduce the multiplication operators. In this example, the

built-in knowledge about integration leads to the evaluation of the indefinite integral $\int x^3\, e^{2x}\, dx$ as $e^{2x}\left(-\frac{3}{8}+\frac{3x}{4}-\frac{3x^2}{4}+\frac{x^3}{2}\right)$.

15 Computation (Reduction of the Green's Operator as an Analytic Polynomial)

Compute[$(1-\lceil 1-x\rceil L-\lceil x\rceil R)\,A^2$]

$$-A\lceil x\rceil-\lceil x\rceil B+\lceil x\rceil A\lceil x\rceil+\lceil x\rceil B\lceil x\rceil$$

□

16 Computation (Reduction of an Integral Operator with Hybrid Polyexponential Kernel)

Compute[$A\lceil x^3\, e^{2x}\rceil A$]

$$\frac{3}{8}A\lceil e^{2x}\rceil-\frac{3}{4}A\lceil e^{2x}x\rceil+\frac{3}{4}A\lceil e^{2x}x^2\rceil-\frac{1}{2}A\lceil e^{2x}x^3\rceil-\frac{3}{8}\lceil e^{2x}\rceil A+\frac{3}{4}\lceil e^{2x}x\rceil A-\frac{3}{4}\lceil e^{2x}x^2\rceil A+\frac{1}{2}\lceil e^{2x}x^3\rceil A$$

□

The polynomial interaction equalities induce a rewrite system, and our first claim is that it is *noetherian*. This means that the chain of computation steps terminates for any input in $\mathcal{An}(\mathcal{F})$.

17 Theorem (Termination of the Reduction System for Analytic Polynomials)

The reduction system generated by orienting the interaction equalities in Input 14 from left to right is a noetherian relation on $\mathcal{An}(\mathcal{F})$.

Let us first *clarify the signature and the reduction relation.* We have a term rewriting system with a flat and commutative flexible-arity symbol $+$, a flat flexible-arity symbol $\cdot$, a nullary operation for each coefficient in $\mathbb{C}$ as constructor of the corresponding constant polynomial, and a nullary operation for each indeterminate in $\Xi=\{D, A, B, L, R\}\bigcup\{\lceil f\rceil \mid f\in\mathcal{F}^{\#}\}$ as a constructor of the corresponding 'solitary' polynomial. Since $+$ does not occur on the left-hand side of any rule, we need not worry about its flatness nor associativity. We will take care of the associativity of $\cdot$ by applying an associative matcher. Moreover, we have a ground term rewriting system, because f acts only as an external variable (it cannot be instantiated by a polynomial). In other words, there are infinitely many rewrite rules for each rule of Input 14 that contains a multiplication operator $\lceil f\rceil$; every instantiation of f over $\mathcal{F}^{\#}$ gives rise to one rule. Whenever we speak of "a rule", we actually mean the whole family of rules arising from these instantiations.

We will first assume that we do not distinguish basis polynomials, i.e. we will work with the indeterminates $\overline{\Xi}=\{D, A, B, L, R\}\bigcup\{\lceil f\rceil \mid f\in\mathcal{F}\}$ and we do all reduction *without basis expansion*. We must show that there is no infinite chain of such reductions, and we do this by an indirect proof.

So assume there were such an infinite reduction chain $p_1\to p_2\to p_3\to\ldots$ in $\mathbb{C}\langle\overline{\Xi}\rangle$. Each of the rules "DA", "DB", "DL", "DR", "AMD", "BMD", "AD", "BD" of Input 14 decrease the number of occurrences of D in a polynomial (which is a kind of *differential weight* of the corresponding operator), and it is never increased by any rule. Hence there can only be finitely many applications of these rules in the initial part of the chain

$p_1 \to p_2 \to p_3 \to \ldots$; taking this initial part out, we are left with another infinite reduction chain, which we might as well denote by $p_1 \to p_2 \to p_3 \to \ldots$ again, and this chain does not use the rules just mentioned.

Next we consider the number of occurrences of either A or B, so to say the integral weight; the rules "LA", "RA", "AMA", "AMB", "BMA", "BMB", "AA", "AB", "BA", "BB", "AL", "BL", "AR", "BR", "AML", "BML", "AMR", "BMR" all decrease the *integral weight*, which is again never increased by any rule. Hence we may disregard these rules as well, assuming that the reduction chain $p_1 \to p_2 \to p_3 \to \ldots$ applies only the remaining rules. Finally, let us consider the number of occurrences of either L or R, which could be called the boundary weight; the rules "RA", "LB", "LL", "LR", "RL", "RR" all decrease the *boundary weight*, and once again it is never increased by any rule. So let us also disregard these rules in $p_1 \to p_2 \to p_3 \to \ldots$.

The product rule "DM" can be discarded on grounds of positioning: Let $\delta(w)$ denote the position of the leftmost occurrence of D in a word $w \in \overline{\Xi}^*$, and let $\delta(p) = \sum_{w \in \mathrm{supp}(p)} \delta(w)$ be what could be called the *cumulative differential position* of a polynomial p. Obviously, each application of the rule "DM" decreases the cumulative differential position by 1, and this cannot go on forever since δ ranges over $\mathbb{N}$ and is never increased by any other rule. A similar argument based on a 'cumulative boundary position' allows to discard the rules "LM" and "RM".

Hence we are only left with the rule "MM". But this rule always decrease the *total length* of a polynomial, defined as the sum of the lengths of all words in its support. Hence it can also not be applied infinitely often, and the original assumption that we had an infinite reduction chain is falsified.

For concluding the proof, we must show that basis expansion does not spoil anything; but this is trivial. Assume we had an infinite reduction $p_1 \to p_2 \to p_3 \to \ldots$, this time in $\mathbb{C}\langle\Xi\rangle$. Then some of the reduction steps will be reductions in $\mathbb{C}\langle\overline{\Xi}\rangle$, whereas the rest are basis expansions. The former is finite by what we have proved above, whereas the latter is finite as well because basis expansion is clearly noetherian.

Let us also point out an alternative, more algebraic way of proving the noetherianity of the reduction on $\mathbb{C}\langle\overline{\Xi}\rangle$; I learned this proof from Ralf Hemmecke. Let $\Omega = \{D, A, B, L, R, M\}$ the set of 'reduced indeterminates', containing only one 'generic' multiplication operator M. We order Ω by $D > A > B > L > R > M$ and the word monoid Ω^* by the induced graded lexicographic ordering, also denoted by $>$. The ordering $>$ is clearly total and noetherian on Ω^*. Moreover, it is easily verified that $>$ is monotonic with respect to the multiplication (being the concatenation of words) on Ω^*, so have got a well-ordered monoid.

Using Theorem 4.69 of [3], the ordering on Ω^* induces a well-ordering on its finite subsets $\mathcal{P}_{\mathrm{fin}}(\Omega^*)$ defined recursively by $A \succeq \emptyset$ for all $A \in \mathcal{P}_{\mathrm{fin}}(\Omega^*)$ and

$$\emptyset \not\succeq B,$$
$$A \succeq B \;\Leftrightarrow\; \max_> A > \max_> B \vee (\max_> A = \max_> B \wedge A' \succeq B')$$

for all $A, B \in \mathcal{P}_{\mathrm{fin}}(\Omega^*)\backslash\{\emptyset\}$. Here we have written S' for $S\backslash\{\max_> S\}$, with S being any element of $\mathcal{P}_{\mathrm{fin}}(\Omega^*)\backslash\{\emptyset\}$. As usual, the strict part of $\succeq$ will be denoted by $\succ$.

Then we define a mapping $v : \mathbb{C}\langle\overline{\Xi}\rangle \to \mathcal{P}_{\text{fin}}(\Omega^*)$ by $v(p) = \text{supp}(\tilde{p})$, where supp is the polynomial support and $\tilde{} : \mathbb{C}\langle\overline{\Xi}\rangle \to \mathbb{C}\langle\Omega\rangle$ is the homomorphic extension of the map $\overline{\Xi} \to \Omega$ sending every $\lceil f \rceil$ to M and fixing all other elements. This gives rise to a strict partial order on $\mathbb{C}\langle\overline{\Xi}\rangle$ defined by $p \gg q \Leftrightarrow v(p) \succ v(q)$. By the well-orderedness of $\succeq$, the new ordering $\gg$ is noetherian. Therefore it suffices to prove that all the reductions are compatible with $\gg$ in the sense that $p \to p'$ implies $p \gg p'$ for all $p, p' \in \mathbb{C}\langle\overline{\Xi}\rangle$.

Looking the reduction rules of Input 14, we can easily verify that they respect the ordering $\gg$, meaning that for every rule $w = p$ one has $w \gg p$, where $w \in \overline{\Xi}^*$ and $p \in \mathbb{C}\langle\overline{\Xi}\rangle$. This immediately carries over to reductions. Assume we have

$$p_1 + \lambda\, w_1\, w\, w_2 + p_2 \to p_1 + \lambda\, w_1\, p\, w_2 + p_2,$$

via the rule $w = p$, where $p_1, p_2, p \in \mathbb{C}\langle\overline{\Xi}\rangle$ and $w_1, w_2, w \in \overline{\Xi}^*$ and $\lambda \in \mathbb{C}^\times$ such that $v_1 > w_1\, w\, w_2 > v_2$ for all $v_1 \in \text{supp}(p_1)$ and $v_2 \in \text{supp}(p_2)$. The argument to be used here is closely analogous to Lemma 5.20(iv) of [3], so let us only sketch the idea: Since all the words in p_1 remain untouched, it suffices to show $\lambda\, w_1\, w\, w_2 + p_2 \gg \lambda\, w_1\, p\, w_2 + p_2$. But this is clear because all the words in $\lambda\, w_1\, p\, w_2$ and p_2 are smaller than $w_1\, w\, w_2$ with respect to $>$, and some of them may even cancel out.

□

The crucial fact about the reduction system of Input 14 is that we can use it for computing the Green's function. This is guaranteed by the fact that the *normal forms* (and we have just proved that a normal form always exists!) are of an appropriate shape, as we will show now. In the next step, we will prove that the normal forms are even canonical, but note that we do not really need this for the application of computing Green's functions. It is a kind of mathematical luxury, implying that we are working in a "beautiful" structure that may be characterized as a suitable quotient via Proposition 31.

18 Definition (Normal Form of Analytic Polynomials)

A polynomial of $\mathcal{An}(\mathcal{F})$ is said to be in *normal form* iff all its monomials are produced by the rule for $\mathcal{M}$ in the following grammar:

Production Rule	*Name*
$\mathcal{M} :=$ $\mathcal{AIA} \mid \mathcal{AD} \mid \mathcal{ABD}$	Monomial Operator
$\mathcal{I} := A \mid B$	Integral Operator
$\mathcal{A} := 1 \mid \lceil f \rceil$	Algebraic Operator
$\mathcal{B} := L \mid R$	Boundary Operator
$\mathcal{D} := 1 \mid D\mathcal{D}$	Differential Operator

We denote the set of these normal forms by $\mathcal{Gr}(\mathcal{F})$, and we call them the *Green's polynomials* over $\mathcal{F}$.

□

Observe that they really provide a *solution to the problem of finding Green's functions* (see Theorem 50 for the precise proof): Assuming that we have somehow computed an analytic polyno-

mial representing the Green's operator for a given BVP (and in Definition 47 we will show how to do this algorithmically), the resulting normal form G will consist only of monomials of type $\mathcal{AIA}$ since it is clear that G cannot involve differential or boundary operators but must contain integration operators. As explained before, the terms where $\mathcal{I}$ is A give one branch of the corresponding Green's function, those where $\mathcal{I}$ is B give the other branch; the second $\mathcal{A}$ in $\mathcal{AIA}$ involves the bound variable of the integral quantifier (written as ξ in Equation 4), whereas the first $\mathcal{A}$ involves the free variable (written as x).

19 Theorem (Normal Form of Analytic Polynomials)

The normal forms with respect to the reduction system induced by Input 14 on $\mathcal{An}(\mathcal{F})$ are precisely the Green's polynomials $\mathcal{Gr}(\mathcal{F})$.

It is clearly sufficient to consider only the words (the monomials without the coefficient) because a polynomial is irreducible iff all its words are.

First assume $p \in \mathcal{Gr}(\mathcal{F})$; we must show that p is irreducible. Consider the case $\mathcal{AIA}$: There is no rule for reducing any of the polynomials $\lceil f \rceil A \lceil g \rceil$, $\lceil f \rceil B \lceil g \rceil$, $A \lceil g \rceil$, $B \lceil g \rceil$, $\lceil f \rceil A$, $\lceil f \rceil B$, A, B. Studying the case $\mathcal{AD}$, it suffices to observe that there is no rule for reducing either of $\lceil f \rceil D$, $\lceil f \rceil$, D, 1. For treating the case $\mathcal{ABD}$, we need only consider the polynomials $\lceil f \rceil L D$, $\lceil f \rceil R D$, $L D$, $R D$, $\lceil f \rceil L$, $\lceil f \rceil R$, L, R, and once again there is no rule for reducing them. Hence we know that $\mathcal{Gr}(\mathcal{F})$ does indeed contain only normal forms.

Now assume we have a noncommutative polynomial $p \notin \mathcal{Gr}(\mathcal{F})$; we must show that p is reducible. Let us first observe that $p \neq 1$ since $1 \in \mathcal{Gr}(\mathcal{F})$. We will now proceed by case distinction on the initial letter of p, ranging over the possible indeterminates $\{\mathrm{D}, \mathrm{A}, \mathrm{B}, \mathrm{L}, \mathrm{R}\} \bigcup \{\lceil \mathrm{f} \rceil \mid \mathrm{f} \in \mathcal{F}^{\#}\}$. This proof will realize the intuition behind the reduction process, as it was outlined before.

Assume the first letter of p is D, say $p = D\,p'$. Then there must be some first letter $\lambda \neq D$ in p'; otherwise p would a Green's polynomial of type $\mathcal{D}$. Hence $D\lambda$ occurs as a subword of p, and this subword may be one of $D A$, $D B$, $D L$, $D R$, $D \lceil f \rceil$; all of these are reducible by the rules with the same name (subject to the natural convention that the indeterminates for multiplication operators are represented by an "M"). Observe that all of these rules belong to the "first phase".

Assume the first letter of p is a boundary operator $\mathcal{B}$, say $p = \mathcal{B}\,p'$. If p' were 1, the given word p would be a Green's polynomial of type $\mathcal{ABD}$; so we know that $p' \neq 1$. Furthermore, we may assume that p' does not start with D. For if it does, by what we have proved before, p' is either reducible or in $\mathcal{Gr}(\mathcal{F})$. In the first case, p is reducible as well. In the second case, p' can only be of type $\mathcal{D}$ and hence p of type $\mathcal{ABD}$, contradicting the assumption $p \notin \mathcal{Gr}(\mathcal{F})$. Hence p starts with one of $L A$, $L B$, $L L$, $L R$, $L \lceil f \rceil$ $R A$, $R B$, $R L$, $R R$, $R \lceil f \rceil$; and they can all be reduced by rules with corresponding names. Observe again that these rules belong to the "second phase" of reduction.

Assume the first letter of p is an integral operator $\mathcal{I}$, say $p = \mathcal{I}\,p'$. Again p' must start with some other letter λ; otherwise p would be a Green's polynomial of type $\mathcal{AIA}$. Let us first assume that λ is no $\lceil f \rceil$. Then the first two letters of p are one of $A D$, $A A$, $A B$, $A L$, $A R$, $B D$, $B A$, $B B$, $B L$, $B R$, all of which are reducible by the rules with

corresponding names. Now assume that λ is an $\lceil f \rceil$. In this case, it must be followed by a letter λ'; otherwise $p = \mathcal{I}\lceil f \rceil$ would again be a Green's polynomial of type $\mathcal{AIA}$. If λ' is another $\lceil g \rceil$, we can reduce $\mathcal{I}\lceil f \rceil\lceil g \rceil$ and hence p by the rule "MM". In the remaining cases, the first three letters of p must be among $A\lceil f \rceil D$, $A\lceil f \rceil A$, $A\lceil f \rceil B$, $A\lceil f \rceil L$, $A\lceil f \rceil R$, $B\lceil f \rceil D$, $B\lceil f \rceil A$, $B\lceil f \rceil B$, $B\lceil f \rceil L$, $B\lceil f \rceil R$; and once again they can all be reduced by the rules with corresponding names.

Finally, assume the first letter of p is an $\lceil f \rceil$, say $p = \lceil f \rceil p'$. If p' is 1, again p would be a Green's polynomial, namely of type $\mathcal{AD}$. Hence p' starts with some first letter λ. If λ is D, we know from before that p' (and hence p) is either reducible or a Green's polynomial, which can only be of type $\mathcal{D}$. But in the latter case, p would be a Green's polynomial of type $\mathcal{AD}$, contradicting the assumption on p. If λ is a $\mathcal{B}$, we may again assume that p' is a Green's polynomial. In this case, p' can only be of the form $\mathcal{BD}$, so p would be a Green's polynomial of type $\mathcal{ABD}$, once again contradicting the assumption on p. If λ is an $\mathcal{I}$, we may assume as before that p' is a Green's polynomial. This time we can infer that p' is then of the form $\mathcal{AIA}$, so it is either $\lceil g \rceil\mathcal{IA}$ or $\mathcal{IA}$. Accordingly, p is either $\lceil f \rceil\lceil g \rceil\mathcal{IA}$ or $\lceil f \rceil\mathcal{IA}$. In the former case, we may use the reduction rule "MM", whereas the latter is not possible because it means that p is a Green's polynomial of type $\mathcal{AIA}$. So the only remaining case it that λ is another $\lceil g \rceil$, and hence p starts with the $\lceil f \rceil\lceil g \rceil$, which is of course reducible by the rule "MM" again.

This concludes the proof that all normal forms are Green's polynomials. As we have also proved the converse, this means that the normal forms and $\mathcal{Gr}(\mathcal{F})$ actually coincide, as was claimed.

□

As announced before, the normal forms of the reduction system in Input 14 are even unique. For proving this, it is clearly sufficient to prove that the given reduction system is confluent; see [1]. So we must show that whenever a reduction splits in two possible paths $t_1 \leftarrow t \rightarrow t_2$, the resulting terms t_1 and t_2 have a common successor; they "flow together". Looking at equations of Input 14, we can see that many of them depend heavily on the algebraic structure of $\mathcal{F}$. Therefore we should expect that the confluence proof must have recourse to the axioms of analytic algebras in Definition 11; a few first attempts will immediately confirm this expectation.

As a consequence, we will need two different kinds of computation for establishing confluence: on the one hand, the reductions of Input 14; on the other hand, the axioms of Definition 11. Whereas the former are already oriented (they are "rewrite rules" rather than plain "equalities"), the latter might have to be used in both directions. For a computer-generated confluence proof, though, we would prefer a completely deterministic procedure. Hence we will try to replace the axioms of analytic algebras suitable rewrite rules. In fact, one can find these rules quite easily by just trying out the confluence proof until it gets stuck ("lazy thinking paradigm"). Let us start with some first results about resolving combinations of integral and boundary actions.

20 Lemma (Boundary Integrals)

Let $\mathcal{F}$ be an analytic algebra. Then the equalities listed below are fulfilled.

Lemma[*"Boundary Integrals"*, any[f, g],

$$\begin{array}{ll} \left(\int^* f\right)^{\leftarrow} = 0 & \text{"la"} \\ \left(\int^* f\right)^{\rightarrow} = \int^* f + \int_* f & \text{"ra"} \\ \left(\int_* f\right)^{\leftarrow} = \int^* f + \int_* f & \text{"lb"} \\ \left(\int_* f\right)^{\rightarrow} = 0 & \text{"rb"} \end{array}\ ]$$

Axiom "ad" allows to eliminate the left boundary action, giving $f^{\leftarrow} = \int^* f' - f$. Substituting this in the left-hand side of equality "la", we get $\int^* (\int^* f)' - \int^* f = \int^* f - \int^* f = 0$ as claimed, the first equality coming from axiom "da". For deducing equality "ra", we eliminate the right boundary action by axiom "bd", yielding $f^{\rightarrow} = \int_* f' + f$. Again, this is substituted in the left-hand side of equality "ra", and we obtain $\int_* (\int^* f)' + \int^* f = \int_* f + \int^* f = \int^* f + \int_* f$ as claimed, using again axiom "da" in the first equality. Equalities "lb" and "rb" are deduced analogously.

□

The next group of results deals with combinations among the integration actions. It will be convenient to introduce a parsing convention for iterated integrals in order to avoid some parentheses.

21 Convention (Precedence of Integral Operators)

The term $\int^* S\,T$ is to be parsed as $\int^* (S\,T)$ rather than $(\int^* S)\,T$, where S and T are arbitrary terms. Similar conventions apply for the other integral operators.

□

Let us now state the equalities describing the essential interactions between the integration actions (compare them to the interaction rules "AMA", "AMB", "BMA", "BMB"). Basically, these equalities express partial integration in analytic algebras.

22 Lemma (Integration Laws)

Let $\mathcal{F}$ be an analytic algebra. Then the equalities listed below are fulfilled.

Lemma[*"Integration Laws"*, any[f, g],

$$\begin{array}{ll} \int^* f \left(\int^* g\right) + \int^* g \left(\int^* f\right) = \left(\int^* f\right)\left(\int^* g\right) & \text{"a:a"} \\ \int^* f \left(\int_* g\right) - \int^* g \left(\int^* f\right) = \left(\int^* f\right)\left(\int_* g\right) & \text{"a:b"} \\ \int_* f \left(\int^* g\right) - \int_* g \left(\int_* f\right) = \left(\int_* f\right)\left(\int^* g\right) & \text{"b:a"} \\ \int_* f \left(\int_* g\right) + \int_* g \left(\int_* f\right) = \left(\int_* f\right)\left(\int_* g\right) & \text{"b:b"} \end{array}\ ]$$

Equality "a:a" follows from the product rule of differentiation: Substituting $\int^* f$ for F and $\int^* g$ for G in $F'\,G + G'\,F = (F\,G)'$ yields $f \int^* g + g \int^* f = ((\int^* f)\,(\int^* g))'$ upon using axiom "da". Applying $\int^*$ on this equality yields "a:a", using axioms "ad", "lm" and equality "la" for simplifying the right-hand side. The other equalities are deduced analogously.

□

Looking at the axioms of analytic algebras and the equalities derived up to now, we can see a certain symmetry between the integral and cointegral actions (as operators in the Hilbert space $L^2[a, b]$, they are in fact duals of each other). Such a symmetry is nice for axiomatization and abstract proofs but it does suitable for rewriting. Using these symmetries, we will eliminate one in favor of the other. In order to do so, we will now introduce the definite integral as the crucial link between integral and cointegral action.

23 Definition (Definite Integral)

Let $\mathcal{F}$ be an analytic algebra and fix an arbitrary $f \in \mathcal{F}$. Then $\int^* f + \int_* f$ is called the *definite integral* of f and is denoted by $\oint f$.

□

Now we can establish the essential properties of the definite integral. In the next step, we will have recourse to these properties for eliminating the cointegral in favor of the integral action (but we might as well do the opposite).

24 Lemma (Properties of the Definite Integral)

Let $\mathcal{F}$ be an analytic algebra. Then the definite integral is a constant, and the equalities listed below are fulfilled.

Lemma[*"Properties of the Definite Integral"*, any[f, g],

$\oint f' = f^{\rightarrow} - f^{\leftarrow}$	"cd"
$\oint f = \left(\int^* f\right)^{\rightarrow}$	"ra"
$\oint f \left(\int^* g\right) + \oint g \left(\int^* f\right) = (\oint f)(\oint g)$	"c:a"
$\oint f \left(\int_* g\right) + \oint g \left(\int_* f\right) = (\oint f)(\oint g)$	"c:b"

]

Equality "cd" follows by summing axioms "ad" and "bd", and equality "ra" as in Lemma 20, just using the definition of the definite integral. Note that equality "ra" implies that definite integrals are constant by the axiom "lr:".

Summing equalities "a:a" and "b:a" yields for $\oint f(\int^* g)$ the term $(\oint f)(\int^* g) + \int_* g\left(\int_* f\right) - \int^* g(\int^* f)$. Replacing $\int_* f$ by $\oint f - \int^* f$ in this term yields $(\oint f)(\oint g) - \oint g(\int^* f)$, using the fact that definite integrals are constant. Hence we have $\oint f(\int^* g) = (\oint f)(\oint g) - \oint g(\int^* f)$, which is equivalent to equality "c:a". By an analogous deduction, one can derive equality "c:b".

□

Finally, we can formulate all the rewrite rules to be used in the confluence proof (we write them as equalities again, but of course they are understood as oriented from left to right). We have split them in three groups. Let us start with the first group treating the basic interaction laws between the action operators.

25 Input (Interaction Laws)

Assumptions[*"Interaction Laws"*, any[f, g],

$(f\,g)' = f'\,g + f\,g'$	"dm"
$(f^2)' = 2\,f\,f'$	"ds"
$\left(\int^* f\right)' = f$	"da"
$\int^* f' = f - f^{\leftarrow}$	"ad"
$\oint f' = f^{\rightarrow} - f^{\leftarrow}$	"cd"]
$\left(\int^* f\right)^{\leftarrow} = 0$	"la"
$\left(\int^* f\right)^{\rightarrow} = \oint f$	"ra"
$\int_* f = \oint f - \int^* f$	"b"

The rules "dm", "da", "ad" are axioms. Rule "ds" is just a special case of rule "dm" (it is used only for making the computer-generated reduction process easier to implement). Rule "cd" was derived in Lemma 24, rules "la" and "ra" in Lemma 20. Finally, rule "b" is simply the definition of the definite integral $\oint$, now used for eliminating the cointegral in favor of the integral action.

□

The next group of rewrite rules deals with integration. They are basically "eliminated" versions of the integration laws of Lemma 22.

26 Input (Integration Laws)

Assumptions[*"Integration Laws"*, any[f],

$\int^* \int^* 1 = \frac{1}{2}\left(\int^* 1\right)^2$	"aa1"
$\oint \int^* 1 = \frac{1}{2}\left(\oint 1\right)^2$	"ca1"
$\int^* f \left(\int^* f\right) = \frac{1}{2}\left(\int^* f\right)^2$	"ama"]
$\oint f \left(\int^* f\right) = \frac{1}{2}\left(\oint f\right)^2$	"cma"
$\int^* \int^* f = \left(\int^* 1\right)\left(\int^* f\right) - \int^* \left(\int^* 1\right) f$	"aa"
$\oint \int^* f = \left(\oint 1\right)\left(\oint f\right) - \oint \left(\int^* 1\right) f$	"ca"

Rules "ama" and "cma" follow immediately from rule "a:a" in Lemma 22 and rule "c:a" in Lemma 24, respectively, by substituting f for both f and g. Rules "aa" and "ca" follow in the same way by substituting 1 for f and f for g. Rules "aa1" and "ca1" are just trivial special cases of rules "aa" and "ca", respectively, just added for ease of implementation.

□

The last group of rewrite rules are essentially repetitions of axioms.

27 Input (Boundary Laws)

Assumptions[*"Boundary Laws"*, any[f, g],

$$\begin{array}{ll} (f\,g)^{\leftarrow} = f^{\leftarrow}\,g^{\leftarrow} & \text{"lm"} \\ (f^2)^{\leftarrow} = (f^{\leftarrow})^2 & \text{"ls"} \\ (f\,g)^{\rightarrow} = f^{\rightarrow}\,g^{\rightarrow} & \text{"rm"} \\ (f^2)^{\rightarrow} = (f^{\rightarrow})^2 & \text{"rs"} \end{array}\Big]$$

Rules "lm" and "rm" are axioms, and their companions "ls" and "rs" trivial special cases, again added for making the implementation easier.

□

Now we are finally ready for the confluence proof itself.

28 Theorem (Confluence of the Reduction System for Analytic Polynomials)

The reduction system generated by orienting the interaction equalities in Input 14 from left to right is confluent.

As for the termination proof, we will first give the proof without considering basis expansion; so let Ξ and $\overline{\Xi}$ be as before. By Lemma 1.2 of [4], it suffices to prove that all ambiguities of the reduction system are resolvable. In general, one has to consider both overlap and inclusion ambiguities. Inspecting Input 14, however, we can see that there are no inclusion ambiguities, so we can concentrate on the overlap ambiguities, i.e. a pair of rules $w\,w_1 \to p_1$ and $w_2\,w \to p_2$ with $w, w_1, w_2 \in \overline{\Xi}^*$ and $p_1, p_2 \in \mathbb{C}\langle\overline{\Xi}\rangle$. We must show that the corresponding S-polynomials $w_2\,p_1 - p_2\,w_1$ reduce to 0.

This is done in Computation 29, which uses the rewrite rules of Inputs 25, 26, 28 derived before. Note that everything is generated automatically, only that we do not have space enough for listing all of the 233 reductions because they cover approximately 2000 lines. But the computation module checks whether they all come out to zero. Since this is the case, we may conclude (trusting the implementation or reading 2000 lines of proof!) that the given reduction system is indeed confluent.

□

29 Computation (S-Polynomial Reduction)

ProveConfluence[]

The rules DA and AMA yield the S-polynomial:

$$\lceil f\rceil\,\mathrm{A} - \mathrm{D}\lceil\textstyle\int^* f\rceil\,\mathrm{A} + \mathrm{D}\,\mathrm{A}\lceil\textstyle\int^* f\rceil \overset{(\ldots)}{\downarrow} =$$

$$\lceil f\rceil\,\mathrm{A} - \mathrm{D}\lceil\textstyle\int^* f\rceil\,\mathrm{A} + \boxed{\mathrm{D}\,\mathrm{A}}\lceil\textstyle\int^* f\rceil \overset{(DA)}{\downarrow} =$$

$$\lceil\textstyle\int^* f\rceil + \lceil f\rceil\,\mathrm{A} - \boxed{\mathrm{D}\lceil\textstyle\int^* f\rceil}\,\mathrm{A} \overset{(DM)}{\downarrow} =$$

$$\lceil\textstyle\int^* f\rceil + \lceil f\rceil\,\mathrm{A} - \left\lceil\boxed{(\textstyle\int^* f)'}\right\rceil\mathrm{A} - \lceil\textstyle\int^* f\rceil\,\mathrm{D}\,\mathrm{A} \overset{(da)}{\downarrow} =$$

$$\lceil \int^* f \rceil - \lceil \int^* f \rceil \boxed{\mathrm{D\,A}} \overset{(\mathrm{DA})}{\underset{\downarrow}{=}}$$

$$0 \qquad \square$$

$$\ldots$$

The rules RA and AMA yield the S-polynomial:

$$\mathrm{A}\lceil f \rceil \mathrm{A} + \mathrm{B}\lceil f \rceil \mathrm{A} - \mathrm{R}\left\lceil \int^* f \right\rceil \mathrm{A} + \mathrm{R\,A}\left\lceil \int^* f \right\rceil \overset{(\ldots)}{\underset{\downarrow}{=}}$$

$$\mathrm{A}\lceil f \rceil \mathrm{A} + \mathrm{B}\lceil f \rceil \mathrm{A} - \mathrm{R}\left\lceil \int^* f \right\rceil \mathrm{A} + \boxed{\mathrm{R\,A}}\left\lceil \int^* f \right\rceil \overset{(\mathrm{RA})}{\underset{\downarrow}{=}}$$

$$\mathrm{A}\left\lceil \int^* f \right\rceil + \mathrm{B}\left\lceil \int^* f \right\rceil + \mathrm{A}\lceil f \rceil \mathrm{A} + \mathrm{B}\lceil f \rceil \mathrm{A} - \boxed{\mathrm{R}\left\lceil \int^* f \right\rceil}\mathrm{A} \overset{(\mathrm{RM})}{\underset{\downarrow}{=}}$$

$$\mathrm{A}\left\lceil \int^* f \right\rceil + \mathrm{B}\left\lceil \int^* f \right\rceil - \boxed{\left(\int^* f\right)^{\rightarrow}}\mathrm{R\,A} + \mathrm{A}\lceil f \rceil \mathrm{A} + \mathrm{B}\lceil f \rceil \mathrm{A} \overset{(\mathrm{ra})}{\underset{\downarrow}{=}}$$

$$\mathrm{A}\left\lceil \int^* f \right\rceil + \mathrm{B}\left\lceil \int^* f \right\rceil - (\oint f)\boxed{\mathrm{R\,A}} + \mathrm{A}\lceil f \rceil \mathrm{A} + \mathrm{B}\lceil f \rceil \mathrm{A} \overset{(\mathrm{RA})}{\underset{\downarrow}{=}}$$

$$-(\oint f)\,\mathrm{A} - (\oint f)\,\mathrm{B} + \mathrm{A}\left\lceil \int^* f \right\rceil + \mathrm{B}\left\lceil \int^* f \right\rceil + \boxed{\mathrm{A}\lceil f \rceil \mathrm{A}} + \mathrm{B}\lceil f \rceil \mathrm{A} \overset{(\mathrm{AMA})}{\underset{\downarrow}{=}}$$

$$-(\oint f)\,\mathrm{A} - (\oint f)\,\mathrm{B} + \mathrm{B}\left\lceil \int^* f \right\rceil + \left\lceil \int^* f \right\rceil \mathrm{A} + \boxed{\mathrm{B}\lceil f \rceil \mathrm{A}} \overset{(\mathrm{BMA})}{\underset{\downarrow}{=}}$$

$$-(\oint f)\,\mathrm{A} - (\oint f)\,\mathrm{B} + \mathrm{B}\left\lceil \boxed{\int_* f} \right\rceil + \left\lceil \int^* f \right\rceil \mathrm{A} + \left\lceil \boxed{\int_* f} \right\rceil \mathrm{A} \overset{(\mathrm{b})}{\underset{\downarrow}{=}}$$

$$0 \qquad \square$$

$$\ldots$$

The rules BR and RR yield the S-polynomial:

$$-\mathrm{B\,R} + \left\lceil \int_* 1 \right\rceil \mathrm{R}^2 \overset{(\ldots)}{\underset{\downarrow}{=}}$$

$$-\mathrm{B\,R} + \left\lceil \boxed{\int_* 1} \right\rceil \mathrm{R}^2 \overset{(\mathrm{b})}{\underset{\downarrow}{=}}$$

$$-\mathrm{B\,R} + (\oint 1)\boxed{\mathrm{R}^2} - \left\lceil \int^* 1 \right\rceil \mathrm{R}^2 \overset{(\mathrm{RR})}{\underset{\downarrow}{=}}$$

$$(\oint 1)\,\mathrm{R} - \mathrm{B\,R} - \left\lceil \int^* 1 \right\rceil \boxed{\mathrm{R}^2} \overset{(\mathrm{RR})}{\underset{\downarrow}{=}}$$

$$(\oint 1)\,\mathrm{R} - \boxed{\mathrm{B\,R}} - \left\lceil \int^* 1 \right\rceil \mathrm{R} \overset{(\mathrm{BR})}{\underset{\downarrow}{=}}$$

$$(\oint 1)\,\mathrm{R} - \left\lceil \boxed{\int_* 1} \right\rceil \mathrm{R} - \left\lceil \int^* 1 \right\rceil \mathrm{R} \overset{(\mathrm{b})}{\underset{\downarrow}{=}}$$

0 □

☑ Computed 233 S-polynomials in 129 seconds.
☑ Reduced them in 3144 seconds.
☑ All of them reduced to zero!

□

Note that the action operators are "used" in a quite different way in this confluence proof. In a *concrete* reduction like Computation 16, operators such as $\int^*$ are simply evaluated when used on a function such as e^{2x}. In the *abstract* reductions of Computation 29, however, we are only referring to some properties guaranteed by the axioms.

Having a convergent reduction system on $\mathcal{An}(\mathcal{F})$, we can now *transfer its algebraic structure to the normal forms* $\mathcal{Gr}(\mathcal{F})$ as announced before: This means that the confluence result provides us with an algebraic term model of certain operators on $C^\infty[a, b]$, identifying by the equalities in Input 14 "all" those terms that represent the same operator on $C^\infty[a, b]$. Of course, by "all terms" we mean all those whose identification seems relevant to our present purposes.

30 Definition (Green's Ideal)

Let $\mathcal{F}$ be an analytic algebra. Then $\mathcal{An}_0(\mathcal{F})$ denotes the two-sided ideal of $\mathcal{An}(\mathcal{F})$ generated by the reduction system of Input 14. In other words, $\mathcal{An}_0(\mathcal{F})$ consists of all linear combinations of the polynomials $p\,(l - r)\,q$, where $l = r$ is a rule of the reduction system and p, q are from $\mathcal{Gr}(\mathcal{F})$. We call $\mathcal{An}_0(\mathcal{F})$ the *Green's ideal* over $\mathcal{F}$.

□

Now the fact that the reduction system of Input 14 is confluent can also be expressed in a well-known ring-theoretic language: The corresponding set of polynomials (consisting of all $l - r$ for every rule $l = r$ in Input 14) is a *non-commutative Gröbner basis* for $\mathcal{An}_0(\mathcal{F})$; see Theorem 8 of [66]. This leads us back to our observations after Equation 22, once again emphasizing the central role played by the concept of Gröbner basis. In fact, it is now clear why we could avoid the costly computation of a noncommutative Gröbner basis for the problems considered here: We already *have* one, and it need not be changed for different instances of BVPs because by our construction using right inversion and the nullspace projector, everything boils down to reducing $(1 - P)\,T^\blacklozenge$ with respect to the fixed Gröbner basis; see Input 47 for the final formulation.

It is in this ring-theoretic context that we can formulate the announced result about the *algebraic structure* induced by the identifications made in Input 14.

31 Proposition (Green's Factor Algebra)

Let $\mathcal{F}$ be an analytic algebra. Then the Green's polynomials $\mathcal{Gr}(\mathcal{F})$ constitute an algebra isomorphic to the factor algebra $\mathcal{An}(\mathcal{F})\,/\,\mathcal{An}_0(\mathcal{F})$, which we call the Green's algebra.

By Theorem 1.2 of [4].

□

Having a confluent reduction system, the *ideal membership* problem is also settled. We will not need this result in the context of solving BVP; we mention it here just because it is one of the first questions a ring theorist would ask.

32 Definition (Ideal Membership)

Let $\mathcal{F}$ be an analytic algebra. Two polynomials $f, g \in \mathcal{G}r(\mathcal{F})$ are congruent to each other with respect to the Green's ideal $\mathcal{A}n_0(\mathcal{F})$ iff $f - g \in \mathcal{A}n_0(\mathcal{F})$; this will be denoted by $f \equiv_{\mathcal{F}} g$.

□

33 Proposition (Congruence and Reduction)

Let $\mathcal{F}$ be an analytic algebra. Then we have $f \equiv_{\mathcal{F}} g$ iff $f \overset{*}{\leftrightarrow}_{\mathcal{F}} g$ for any $f, g \in \mathcal{A}n(\mathcal{F})$, where $\rightarrow_{\mathcal{F}}$ denotes the reduction induced by the system of Input 14.

The proof is completely analogous to that of Lemma 5.26 in [3], which refers to the commutative case. The only difference is that for reducing a noncommutative polynomial one has to multiply from both sides. Observe also that we do not need the confluence property enjoyed by the system of Input 14; the statement is true for any (commutative as well as noncommutative) polynomial reduction system.

□

34 Proposition (Ideal Membership)

Let $\mathcal{F}$ be an analytic algebra. Then for any $f \in \mathcal{A}n(\mathcal{F})$, we have $f \in \mathcal{A}n_0(\mathcal{F})$ iff $f \overset{*}{\rightarrow}_{\mathcal{F}} 0$.

If $f \in \mathcal{A}n_0(\mathcal{F})$, then $f \equiv_{\mathcal{F}} 0$ by the definition of ideal congruence. By Proposition 33, this implies $f \overset{*}{\leftrightarrow}_{\mathcal{F}} 0$. But we know from Theorem 28 that $\rightarrow_{\mathcal{F}}$ is confluent and hence Church-Rosser by Theorem 8.1.2 in [70]. This means that f and 0 have a common successor (possibly including themselves). But 0 has no proper successor, so the common successor must be 0 and $f \overset{*}{\rightarrow}_{\mathcal{F}} 0$.

Conversely, assume $f \overset{*}{\rightarrow}_{\mathcal{F}} 0$. Then a fortiori $f \overset{*}{\leftrightarrow}_{\mathcal{F}} 0$ and so by Proposition 33 also $f \equiv_{\mathcal{F}} 0$, which is again equivalent to $f \in \mathcal{A}n_0(\mathcal{F})$ by the definition of ideal membership.

□

We have not yet spoken much about the relation between the algebraic structures $\mathcal{A}n(\mathcal{F})$ and $\mathcal{G}r(\mathcal{F})$ and the 'real' operators modeled by them. Let us first make precise what we mean by this "modeling".

35 Definition (Operator Model)

Let $\mathcal{F}$ be an analytic algebra, $\mathcal{A}$ an algebra containing $\mathcal{F}$, and $\mathcal{L}$ a subalgebra of the algebra of linear operators on $\mathcal{A}$. Given a mapping $i : \{D, A, B, L, R\} \rightarrow \mathcal{L}$, extend it to a mapping $i_{\lceil\rceil}$ on $\{D, A, B, L, R\} \bigcup \{\lceil f \rceil \mid f \in \mathcal{F}\}$ by setting $i_{\lceil\rceil}(\lceil f \rceil)(a) = f\,a$ for all $a \in \mathcal{A}$. Then let I be the homomorphic extension of $i_{\lceil\rceil}$ to all of $\mathcal{A}n(\mathcal{F})$. We call I the *interpretation* induced by i and $\mathcal{L}$ the *operator model* (or briefly model) of $\mathcal{A}n(\mathcal{F})$ under I.

□

Now we can make it clear what we mean by the 'real' operators working on $C^\infty[a, b]$.

36 Definition (Smooth Model)

Let $\mathcal{S}$ be the algebra of all linear operators on the space $C^\infty[a, b]$ of smooth functions on a finite real interval $[a, b]$. More precisely, we regard all operators in the Banach space $C[a, b]$ with Chebyshev norm $\|\cdot\|$, being defined on the dense subset $C^\infty[a, b]$. Now define a mapping $sm : \{D, A, B, L, R\} \to \mathcal{S}$ by giving the usual definitions

$$sm(D) = u \mapsto u',$$

$$sm(A) = u \mapsto \left(x \mapsto \int_a^x u(\xi)\, d\xi\right),$$

$$sm(B) = u \mapsto \left(x \mapsto \int_x^b u(\xi)\, d\xi\right),$$

$$sm(L) = u \mapsto (x \mapsto u(a)),$$

$$sm(R) = u \mapsto (x \mapsto u(b)),$$

where u ranges over $C^\infty[a, b]$ and x ranges over $[a, b]$. Let $\mathcal{S}m$ be the interpretation induced by sm, which we will call the *smooth interpretation;* its image $\mathcal{S}m_*(\mathcal{G}r)$ will be called the *smooth model*.

□

Note that $\mathcal{S}m_*(\mathcal{G}r)$ carries no *topology*. In fact, we will not need any topological notions pertaining to the operator algebras used as models for $\mathcal{G}r(\mathcal{F})$. The function space $C^\infty[a, b]$, though, will be used with the topology induced by the Chebyshev norm $\|\cdot\|_\infty$, thus making it into a Banach algebra. We view $C^\infty[a, b]$ as $\{u \in C^\infty(a, b) \mid \forall_{n\in\mathbb{N}}\ u^{(n)} \in C[a, b]\}$.

The restriction to smooth functions is quite severe, though. In Definition 7, we have imposed the smoothness condition not only on the solution function u (thus considering classical solutions) but also on the forcing function f (which is unnecessary even for classical solutions). Besides this, there is a need for more general solutions. In practical examples coming from physics, one often deals with weak solutions; and for studying notions like the fundamental solution of a differential equation, one even needs singular distributions. Therefore it makes sense to introduce a *distributional model* of $\mathcal{G}r(\mathcal{F})$ that will take care of all these desires.

It is actually amazing to see *how easily one can switch* from the smooth to the distributional setting, which demonstrates the power of the algebraic approach of handling BVPs: The Green's algebra is completely 'ignorant' of any setting we have in mind for them—the only essential thing is that the setting ultimately chosen must respect the Green's equalities in the sense of the upcoming Definition 38.

37 Definition (Distributional Model)

Let $\mathcal{D}$ be the algebra of all linear operators on the space $C_0^{-\infty}[a, b]$ of boundary-valued distributions on a finite real interval $[a, b]$. Define a mapping $dis : \{D, A, B, L, R\} \to \mathcal{D}$ by the corresponding formulae of Definition 36, where u now ranges over $C_0^{-\infty}[a, b]$ and x ranges over $[a, b]$ again; of course, now all of these operations are to be understood in the distributional sense. Let $\mathcal{D}is$ be the interpretation induced by dis, which we will call the *distributional interpretation;* its image $\mathcal{D}is_*(\mathcal{G}r)$ will be called the *distributional model*.

□

For more about distributions, consult pages 86-184 in [63] or [43], all following the standard approach due to Laurent Schwartz. For our purposes, however, it seems more natural to follow the alternative but equivalent approach of Sebastião Silva [62], defining $C^{-\infty}[a, b]$ as an inductive limit along the (up-to-isomorphism) inclusion chain

$$C^{\infty}[a, b] \subseteq \ldots \subseteq C^{2}[a, b] \subseteq C^{1}[a, b] \subseteq C[a, b] \subseteq C^{-1}[a, b] \subseteq C^{-2}[a, b] \subseteq \ldots \subseteq C^{-\infty}[a, b].$$

The only subtle point is the boundary values. For a typical distribution, one cannot speak of its "value". Hence we must define what it means that a distribution $u \in C^{-\infty}[a, b]$ has a value $U \in \mathbb{C}$ at a point $x \in [a, b]$. Using Silva's approach as suggested by [22], every $u \in C^{-\infty}[a, b]$ is $v^{(n)}$ for some $v \in C[a, b]$ and $n \in \mathbb{N}$, and we say that u has the value U at x iff

$$\lim_{\xi \to x} \frac{v(\xi)}{(\xi - x)^n} = \frac{U}{n!},$$

where one must use a left-sided or right-sided limit in case x is a or b, respectively. Following the Schwartz approach, one might also use the condition [52] that

$$\forall_{\Delta \in C^{\infty}[a,b]^{\mathbb{N}}} (\Delta \to \delta_x \Rightarrow u \circ \Delta \to U),$$

where δ_x is the delta distribution concentrated at x (note that the first convergence in this implication takes place in the topology of $C^{-\infty}[a, b]$, whereas the second is the usual notion of convergence for real sequences). The space $C_0^{-\infty}[a, b]$ used in Definition 37 consists of all those $u \in C^{-\infty}[a, b]$ such that for all $n \in \mathbb{N}$ the derivatives $u^{(n)}$ have a value at both a and b.

Now the crucial property about these operator models is that they *respect the equalities* specified in Input 14. Let us make this precise by a definition and two subsequent lemmata.

38 Definition (Faithful Model)

Let $\mathcal{F}$ be an analytic algebra and $I : \mathcal{G}r(\mathcal{F}) \to \mathcal{L}$ an interpretation in some operator model $\mathcal{L}$. Then I is called *faithful* iff $I(l) = I(r)$ for all equations $l = r$ of Input 14; in this case, we also say that the model $\mathcal{L}$ is faithful under the given interpretation I.

□

39 Lemma (Faithfulness of the Smooth Model)

The smooth interpretation $\mathcal{S}m$ as specified in Definition 36 is faithful.

This can be verified by routine calculations. In fact, we have derived the equalities of Input 14 by looking at what happens in $\mathcal{S}m_*(\mathcal{G}r)$, and the arguments needed for the verification are essentially contained in Section 1.

□

40 Lemma (Faithfulness of the Distributional Model)

The distributional interpretation $\mathcal{D}is$ as specified in Definition 37 is faithful.

The arguments used for the smooth case basically carry over to the distributional setting by simple continuity arguments.

□

1.3 Inverting Differential Operators

When we solved the BVP of the *classical example* $G\,D^2 = 1 - P$, it was clear how to "solve" for the unknown Green's operator G. Postmultiplying by A^2, we obtained *the* solution $G = (1 - P)\,A^2$. As there are many other right inverses of D^2 besides the "canonical" A^2, there are many solutions G of $D^2\,G = 1$. When premultiplying them by $1 - P$, however, they must all coincide because we know that the solution of $G\,D^2 = 1 - P$ is unique.

But what can we do if we have a more complicated differential operator like $T = D^2 - 3\,D + 7$ instead of the operator D^2 above? As explained at the beginning of the chapter, we restrict ourselves to linear differential operators with constant coefficients. Now every such differential operator is essentially a polynomial in $\mathbb{C}[x]$, just with D figuring as the indeterminate x. And this is also the key to the solution of our problem: We know that *any polynomial splits into linear factors* over $\mathbb{C}$.

For example, we can write the differential operator above as $T = (D - \alpha)\,(D - \overline{\alpha})$, where $\alpha = \frac{1}{2}\left(3 + i\sqrt{19}\right)$ and $\overline{\alpha}$ is its complex conjugate. So in order to right-invert T we have to solve the equation $S(D - \alpha)\,(D - \overline{\alpha}) = 1$ for S. This is easy as soon as we know how to right-invert a linear differential operator with constant coefficient of order one, because then we can do it in *two stages*. Writing a superscript ◆ for the right inverse, we have first $T^{\blacklozenge}(D - \alpha) = (D - \overline{\alpha})^{\blacklozenge}$ and then $T^{\blacklozenge} = (D - \overline{\alpha})^{\blacklozenge}\,(D - \alpha)^{\blacklozenge}$.

So everything boils down to right-inverting differential operators of the shape $D - \lambda$, where λ is some complex number. But this is almost trivial. A little bit of experimentation leads to $(D - \lambda)^{\blacklozenge} = \lceil e^{\lambda x} \rceil\, A\, \lceil e^{-\lambda x} \rceil$. Therefore we arrive at the following formula for right-inverting an arbitrary linear differential operator with constant coefficients (more precisely, an analytic polynomial containing only D as an indeterminate).

41 Input (Differential-Operator Right Inverse)

Formula[*"Differential-Operator Right Inverse"*, any[T],

$$T^{\blacklozenge} = \prod_{i=1,\ldots,n} \lceil e^{\hat{\lambda}_i\, x} \rceil\, A\, \lceil e^{-\hat{\lambda}_i\, x} \rceil \;\Big|\; [p = \text{poly}[T],\ n = \deg[p],\ \hat{\lambda} = \text{rad}[p]]$$

]

□

Everything in this formula is *Theorema* input, which is used as it is for computation; actually, it is part of the algorithm computing the Green's function for a given BVP. We think that this demonstrates a really beautiful point about the usage of integrated mathematical assistants like *Theorema*: The formula above might as well be written on paper in some specialized analysis textbook, but there it would only be dead text. This is not the case here—we can simply select the cell and press SHIFT-ENTER, and the system will *know* the formula so that we can immediately use it as shown in Computation 42 below.

For fully understanding the meaning of the above formula, some *comments* are in order:

- The construct $T \mid [x_1 = T_1, \ldots, x_n = T_n]$ is a notation for the *substitution quantifier*, usually verbalized as "T where x_1 is T_1, …, and x_n is T_n" in prefix reading and as "let x_1 be T_1, …, and x_n be T_n in T" in postfix reading. Here $x_1, \ldots, x_n$ are variables and $T, T_1, \ldots, T_n$ are terms; of course, T will typically contain free occurrences of the variables $x_1, \ldots, x_n$.

- The functions poly, deg, rad are provided to *Theorema* as *built-ins*; they are implemented in Mathematica and explicitly specified as external functions.
- The function poly is used for computing the *characteristic polynomial* in $\mathbb{C}[x]$. For example, $p = \text{poly}[T] = x^2 - 3\,x + 7$ for the differential operator mentioned above.
- The function deg yields the *degree* of a polynomial in $\mathbb{C}[x]$. In the previous example, this gives $n = \deg[p] = 2$.
- The function rad returns all the *roots* of a polynomial in $\mathbb{C}[x]$, repeating some values in case of multiplicities. The result is represented as a vector. For the example above, we have $\hat{\lambda} = \text{rad}[p] = \langle \alpha, \overline{\alpha} \rangle$ with α and $\overline{\alpha}$ as explained before. For the polynomial $q = (x-1)^2 = x^2 - 2\,x + 1$, however, we obtain $\text{rad}[q] = \langle 1, 1 \rangle$. So the length of $\hat{\lambda}$ is always n.
- Using these auxiliary constructions, the body for the term representing $T^\blacklozenge$ is just the *multi-stage iteration* of the simple formula $(D-\lambda)^\blacklozenge = \lceil e^{\lambda x} \rceil\, A\, \lceil e^{-\lambda x} \rceil$ as explained before (note that $\hat{\lambda}_i$ is the i-th component of the vector $\hat{\lambda}$). Using this formula, it is clear that we have indeed $T\,T^\blacklozenge = 1$ for an arbitrary linear differential operator with constant coefficients.

Let us do some small examples using the above definition. We have made a special evaluator for doing various of computations related with searching the Green's function of a BVP. Hence this program is called the Green's evaluator. Basically it just unfolds definitions (such as the one in Input 41), it does some linear algebra when necessary (see Computation 43), and it performs polynomial reduction (as specified in Input 14). See Chapter 2 for details on the implementation.

42 Computation (Examples of Right Inversion)

$\text{Compute}\left[(D^2)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{False}\}\right]$

A^2

$\text{Compute}\left[(D^2)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{True}\}\right]$

$-A\lceil x \rceil + \lceil x \rceil A$

$\text{Compute}\left[(3\,D^2 + 2\,D - 1)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{False}\}\right]$

$\lceil e^{-x} \rceil\, A \left\lceil e^{\frac{4}{3}x} \right\rceil A \left\lceil e^{-\frac{1}{3}x} \right\rceil$

$\text{Compute}\left[(3\,D^2 + 2\,D - 1)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{True}\}\right]$

$\frac{3}{4} \left\lceil e^{\frac{1}{3}x} \right\rceil A \left\lceil e^{-\frac{1}{3}x} \right\rceil - \frac{3}{4} \lceil e^{-x} \rceil\, A\, \lceil e^{x} \rceil$

$\text{Compute}\left[(D^2 + 2\,D + 1)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{False}\}\right]$

$\lceil e^{-x} \rceil\, A^2\, \lceil e^{x} \rceil$

$\text{Compute}\left[(D^2 + 2\,D + 1)^\blacklozenge, \text{by} \to \text{GreenEvaluator}, \text{EvaluatorOptions} \to \{\text{ReduceAfterwards} \to \text{True}\}\right]$

$\lceil e^{-x}\, x \rceil\, A\, \lceil e^{x} \rceil - \lceil e^{-x} \rceil\, A\, \lceil e^{x}\, x \rceil$

□

1.4 Computing the Nullspace Projector

The only missing link for a fully algorithmic treatment of BVP for linear differential operators with constant coefficients (briefly called "differential operators" in the sequel) is the computation of the nullspace projector. For this purpose, let us generalize the procedure followed around Equation (6). We will restrict our attention to the smooth model $\mathcal{Sm}$, so for any $p \in \mathcal{An}$ we will abbreviate $\mathcal{Sm}(p)$ by $\underline{p}$. Now we are given a BVP as specified in Definition 7, determined by a differential operator $\underline{T}$ of order n together with n boundary operators $\underline{B_1}, \ldots, \underline{B_n}$. (Note that according to their specifications, both differential and boundary operators can always be written as interpretations of analytic polynomials.)

We want to find an analytic polynomial P whose interpretation $\underline{P}$ is a projector onto $\mathcal{N}(\underline{T})$ with counter-image $\{\underline{1 - P}\, v \mid v \in C^\infty[a, b]\}$ fulfilling the boundary conditions induced by $\underline{B_1}, \ldots, \underline{B_n}$. As we have seen in Section 1, the computation of the nullspace projector is basically an interpolation problem leading to some trivial linear algebra in $\mathbb{C}^n$. Hence it makes sense to formulate everything *in terms of vectors and matrices*. For this purpose, we will keep the following convention within this section: All the matrices (including vectors which will be understood as row matrices or column matrices) are written with a hat on them, no matter whether their entries are numbers, functions or operators.

Having a fundamental system $u_1, \ldots, u_n$ for the given differential operator $\underline{T}$, let us write $\hat{u}$ for the "fundamental vector" $(u_1, \ldots, u_n)^\top$. For writing the given boundary operators $\underline{B_1}, \ldots, \underline{B_n}$ in terms of a matrix operator, we introduce the operator-valued vector $\hat{D}_n = (1, D, \ldots, D^{n-1})^\top$. We will call $\hat{D}_n$ the *Wronski operator* because it yields the Wronskian matrix $\hat{w}$ when applied to the fundamental vector, so $\hat{w} = \underline{\hat{D}_n}\, \hat{u}^\top$.

Now the *vector boundary operator* $(\underline{B_1}, \ldots, \underline{B_n})^\top$ can be written as $\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}$ for suitable matrices $\hat{l}, \hat{r} \in \mathbb{R}^{n \times n}$. In fact, using the notation of Definition 7, these matrices are given by

$$\hat{\mathrm{l}} = \begin{pmatrix} \mathrm{p}_{1,n} & \mathrm{p}_{1,n-1} & \cdots & \mathrm{p}_{1,0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{p}_{n,0} & \mathrm{p}_{n,n-1} & \cdots & \mathrm{p}_{n,0} \end{pmatrix}, \tag{40}$$

$$\hat{\mathrm{r}} = \begin{pmatrix} \mathrm{q}_{1,n} & \mathrm{q}_{1,n-1} & \cdots & \mathrm{q}_{1,0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{q}_{n,0} & \mathrm{q}_{n,n-1} & \cdots & \mathrm{q}_{n,0} \end{pmatrix}. \tag{41}$$

We are searching a specific *nullspace projector*, i. e. a linear operator $\underline{P}$ such that $\underline{P}\, v = \sum_{i=1}^n \alpha_i(v)\, u_i$ for all $v \in C^\infty[a, b]$, where the $\alpha_1, \ldots, \alpha_n$ are suitable complex numbers depending on the argument v. Our goal is to ensure $\underline{B_1}(v - \underline{P}\, v) = \ldots = \underline{B_n}(v - \underline{P}\, v) = 0$ for all $v \in C^\infty[a, b]$. Hence we have to make sure that $\left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) \underline{P}\, v = \left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) v$ or

$$\sum\nolimits_{i=1}^n \alpha_i(v) \left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) u_i = \left(\underline{L}\, \hat{l}\, \hat{D}_n + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) v.$$

Collecting the unknown coefficients $a_1(v), \ldots, a_n(v)$ into the vector $\hat{\alpha}(v) = (\alpha_1(v), \ldots, \alpha_n(v))^\top$ and assembling the system matrix $\hat{s}$ with $\left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) u_1, \ldots, \left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) u_n$ as its columns, we are left with the matrix equation $\hat{s}\, \hat{\alpha}(v) = \left(\underline{L}\, \hat{l}\, \underline{\hat{D}_n} + \underline{R}\, \hat{r}\, \underline{\hat{D}_n}\right) v$. Looking a bit closer, one sees that $\hat{s}$ is actually $\hat{l}\, \hat{w}^{\leftarrow} + \hat{r}\, \hat{w}^{\rightarrow}$, where $\hat{w}^{\leftarrow} = \hat{w}_{x \leftarrow a}$ and $\hat{w}^{\rightarrow} = \hat{w}_{x \leftarrow b}$ are the usual boundary

actions on the Wronskian matrix. Substituting $\hat{\alpha}(v) = \left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1}\left(\underline{L}\,\hat{l}\,\underline{\hat{D}_n} + \underline{R}\,\hat{r}\,\underline{\hat{D}_n}\right)v$ into the ansatz $\underline{P}\,v = \sum_{i=1}^{n}\alpha_i(v)\,u_i = \hat{u}^{\top}\,\hat{\alpha}(v)$, we obtain

$$\underline{P}\,v = \hat{u}^{\top}\left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1}\left(\underline{L}\,\hat{l}\,\underline{\hat{D}_n} + \underline{R}\,\hat{r}\,\underline{\hat{D}_n}\right)v.$$

Finally, we can now abstract from the argument v, and we see that $\underline{P}$ is indeed the interpretation of a corresponding analytic polynomial, namely

$$\mathrm{P} = \lceil \hat{\mathrm{u}}^{\top} \rceil \left(\hat{\mathrm{l}}\,\hat{\mathrm{w}}^{\leftarrow} + \hat{\mathrm{r}}\,\hat{\mathrm{w}}^{\rightarrow}\right)^{-1}\left(\mathrm{L}\,\hat{\mathrm{l}}\,\hat{\mathrm{D}}_{\mathrm{n}} + \mathrm{R}\,\hat{\mathrm{r}}\,\hat{\mathrm{D}}_{\mathrm{n}}\right); \tag{42}$$

we call this the *nullspace-projector formula.*

We will now *summarize* this result in a *Theorema* formula that is actually used in the corresponding part of the Green's function computation.

43 Input (Nullspace Projector)

Definition[*"Wronski Operator"*, any[n],

$\hat{D}_n = \langle D^i \mid i = 0, \ldots, n-1 \rangle$

]

Formula[*"Nullspace Projector"*, any[$\hat{w}$, $\hat{l}$, $\hat{r}$],

$\mathrm{Proj}_{\hat{w}}[\hat{l}, \hat{r}] = \lceil \hat{w}_1 \rceil \left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1}\left(L\,\hat{l}\,\hat{D}_n + R\,\hat{r}\,\hat{D}_n\right) \Big|\, [n = \dim[\hat{w}]]$

]

□

A few remarks about this definition:

- The Wronski operator is defined by using the *vector quantifier*. The construct $\langle T \mid i = i_0, \ldots, i_1 \rangle$ generates the vector $(T_{i \leftarrow i_0}, \ldots, T_{i \leftarrow i_1})$, where T is an arbitrary term (usually containing the free occurrences of the variable i), i is some variable, i_0 and i_1 are natural numbers. For example, $\langle i^2 \mid i = 3, \ldots 6 \rangle$ is $(9, 16, 25, 36)$, and $\hat{D}_3 = \langle D^i \mid i = 0, \ldots, 2 \rangle$ is (I, D, D^2).
- The *nullspace projector* is computed just as in Equation 42. The only difference is that we have eliminated the need for the fundamental vector $\hat{u}$ as it can be obtained by taking the first row $\hat{w}_1$ of the Wronski matrix $\hat{w}$. In this way, the only argument needed besides the boundary matrices $\hat{l}$ and $\hat{r}$ is the Wronski matrix $\hat{w}$. It is also used for determining the degree n of the given differential operator T because we know that $\hat{w}$ must be an $n \times n$ matrix; we use the built-in function dim for this purpose. The degree n is needed for specifying the appropriate Wronski operator $\hat{D}_n$.
- We use the *function* Proj for denoting the nullspace projector so computed. We view it as depending directly on the boundary conditions (specified through the arguments $\hat{l}$ and $\hat{r}$) and indirectly on the differential operator (specified through the parameter $\hat{w}$). Having the function Proj available, we can use it in the subsequent formula for computing the Green's operator.

Let us now see some short *examples*, using the Green's evaluator again. In the first example, we will just *recompute the classical nullspace projector* obtained in Equation (7) by an ad-hoc procedure.

44 Computation (Nullspace Projector for Classical Heat Conduction)

KnowledgeBase["Heat-Conduction Classical Boundary Matrices",

$$\hat{l} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\hat{r} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

]

KnowledgeBase["Heat-Conduction Wronskian",

$$\hat{w} = \begin{pmatrix} 1 & x \\ 0 & 1 \end{pmatrix}$$

]

KnowledgeBase["Classical Heat Conduction",
  KnowledgeBase["Heat-Conduction Classical Boundary Matrices"]
  KnowledgeBase["Heat-Conduction Wronskian"]
]

Compute[$\text{Proj}_{\hat{w}}[\hat{l}, \hat{r}]$,
  by → GreenEvaluator,
  using → KnowledgeBase["Classical Heat Conduction"]]

$L - \lceil x \rceil L + \lceil x \rceil R$

□

Now let us make sure that we can also impose *initial conditions* instead of boundary conditions, and we can of course prescribe values for the function as well as its derivative in so-called *hybrid conditions*.

45 Computation (Nullspace Projector for Modified Boundary Conditions)

KnowledgeBase["Heat-Conduction Hybrid Boundary Matrices",

$$\hat{l} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

$$\hat{r} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

]

KnowledgeBase["Hybrid Heat Conduction",
  KnowledgeBase["Heat-Conduction Hybrid Boundary Matrices"]
  KnowledgeBase["Heat-Conduction Wronskian"]
]

KnowledgeBase["Heat-Conduction Initial Boundary Matrices",

$$\hat{l} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\hat{r} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

]

```
KnowledgeBase["Initial Heat Conduction",
  KnowledgeBase["Heat-Conduction Initial Boundary Matrices"]
  KnowledgeBase["Heat-Conduction Wronskian"]
]
Compute[Proj_ŵ[l̂, r̂],
  by → GreenEvaluator,
  using → KnowledgeBase["Hybrid Heat Conduction"]]
```

$$R - L\,D + \lceil x \rceil\, L\,D$$

```
Compute[Proj_ŵ[l̂, r̂],
  by → GreenEvaluator,
  using → KnowledgeBase["Initial Heat Conduction"]]
```

$$L + \lceil x \rceil\, L\,D$$

□

Finally let us do a slightly more complicated computation, taken from Example 2 in Krall's book [40] on page 109. It describes damped oscillations, and we will return to this example later for computing its Greens' function.

46 Computation (Nullspace Projector for Damped Oscillations)

```
KnowledgeBase["Damped-Oscillations Wronskian",
  ŵ = ( e^-x      x e^-x
       -e^-x   e^-x - x e^-x )
]
KnowledgeBase["Damped Oscillations",
  KnowledgeBase["Heat-Conduction Classical Boundary Matrices"]
  KnowledgeBase["Damped-Oscillations Wronskian"]
]
Compute[Proj_ŵ[l̂, r̂],
  by → GreenEvaluator, EvaluatorOptions → {BoundaryPoints → {0, π}},
  using → KnowledgeBase["Damped Oscillations"]]
```

$$\lceil e^{-x} \rceil\, L - \pi^{-1}\, \lceil e^{-x}\, x \rceil\, L + (e^{\pi}\, \pi^{-1})\, \lceil e^{-x}\, x \rceil\, R$$

□

1.5 Finding the Green's Operator

We are now approaching the summit of this chapter and the whole thesis. We have assembled all the components for computing the Green's operator of an arbitrary linear differential operator with constant coefficients, so we just need to *put them together*. This is done in the following *Theorema* formula, which is immediately executable for computations as we will demonstrate soon.

47 Input (Green's Operator)

```
Formula["Green Operator", any[T, B],
  Green[T, B] = (1 - P) T◆ | [ŵ = wron[T], l̂ = left[T, B], r̂ = right[T, B], P = Proj_ŵ[l̂, r̂]]
]
```

□

Again we add a few *comments* on this formula:

- We refer to three *built-in functions* wron, left, and right for computing the Wronski matrix, the left and the right boundary matrix respectively. The latter two are rather trivial rearrangements of the coefficients occurring in the boundary operators $B = \langle B_1, \ldots, B_n \rangle$ as explained in Equations (40) and (41); the former uses Mathematica's function DSolve for solving differential equations.
- Having available the Wronski matrix $\hat{w}$ and the left and right boundary matrices $\hat{l}$ and $\hat{r}$, we use the formula of Input 43 for computing the corresponding *nullspace projector* P.
- Furthermore, we use the formula of Input 41 for computing the according *right inverse* $T^\blacklozenge$ of the given differential operator T.
- Finally, the polynomial for the corresponding *Green's operator* is simply computed as $(1 - P)\, T^\blacklozenge$. We denote it by Green$[T, B]$, meaning the Green's operator for the differential operator T and the boundary operators in B.

Let us try out how it works: Now we can do the *classical example* treated in an ad-hoc manner in Section 1 in a very general context, and everything works in one stroke.

48 Computation (Classical Heat Conduction)

```
Compute[Green[D^2, ⟨L, R⟩],
  by → GreenEvaluator]
```

$$-A\lceil x\rceil - \lceil x\rceil B + \lceil x\rceil A\lceil x\rceil + \lceil x\rceil B\lceil x\rceil$$

□

Of course we can do the same with any other BVP that is in the scope outlined at the beginning of this chapter. In particular, we can now do the whole computation for the example with the *damped oscillations* in Krall's book [40] on page 109.

49 Computation (Damped Oscillations)

```
Compute[Green[D^2 + 2 D + 1, ⟨L, R⟩],
  by → GreenEvaluator, EvaluatorOptions → {BoundaryPoints → {0, π}}]
```

$$-\lceil e^{-x}\rceil A\lceil e^{x}\, x\rceil + \pi^{-1}\lceil e^{-x}\, x\rceil A\lceil e^{x}\, x\rceil - \lceil e^{-x}\, x\rceil B\lceil e^{x}\rceil + \pi^{-1}\lceil e^{-x}\, x\rceil B\lceil e^{x}\, x\rceil$$

□

We will conclude this chapter with the most crucial theorem of the whole thesis: the *correctness statement* for our algorithm computing the Green's operator. We formulate it here for the smooth setting introduced at the beginning of Section 2, but one may give a completely analogous proof for the distributional setting, just replacing $C^\infty[a, b]$ by $C^{-\infty}[a, b]$ and $\mathcal{Sm}$ by $\mathcal{Dis}$ and appealing to Lemma 40 instead of Lemma 39.

50 Theorem (Correctness of the Formula for the Green's Operator)

Assume we have a BVP on the real interval $[a, b]$ given by a differential operator $\underline{T}$ and boundary operators $\underline{B_1}, \ldots, \underline{B_n}$, subject to the conditions specified in Definition 7. (For any $p \in \mathcal{A}n$ we abbreviate $\mathcal{S}m(p)$ by $\underline{p}$.)
Let $G \in \mathcal{A}n$ be the result of computing Green$[T, B]$ according to the definition in Inputs 41, 43, 47, and let $G' \in \mathcal{G}r$ be the normal form of G with respect to the reduction system in Input 14. Then G' represents the Green's function for the given BVP.

Computing the analytic polynomial P according to Input 43, we get an operator $\underline{P} : C^\infty[a, b] \to C^\infty[a, b]$ that *projects* everything onto $\mathcal{N}(T)$ by its construction. Since $\underline{T}$ is always surjective, $\underline{1}$ is the only possible projector onto $\mathcal{R}(\underline{T})$. (Note that $\underline{P}$ will usually not be bounded, so we cannot use the Banach-space theory of generalized inverses. In fact, we do not need it, because it is sufficient to work in the naked vector space $C^\infty[a, b]$ with the corresponding Moore-Penrose theory expressed in Definition 1 and Propositions 2, 3; confer the comments made there for more explanations. So $\underline{T}$ and $\underline{B_1}, \ldots, \underline{B_n}$ as well as $\underline{P}$ are plain linear operators.)

By Proposition 2, there is a uniquely determined generalized inverse $\underline{T}^\dagger_{P,1}$, which we will write $\underline{G}$ for some analytic polynomial G yet to be determined. Now $\underline{G}$ is also characterized uniquely by the four corresponding *Moore-Penrose equations,* according to Proposition 3. But as mentioned after Equation 10, the first Moore-Penrose equation is always redundant and the second is as well in our case, due to the trivial range projector. We will now show that the fourth equation also follows from the third equation, as we did for the concrete example in Computation 5. The third equation reads $\underline{GT} = \underline{1 - P}$, which gives $\underline{TGTT^\blacklozenge} = \underline{TT^\blacklozenge - TPT^\blacklozenge}$ upon premultiplying by $\underline{T}$ and postmultiplying by $\underline{T^\blacklozenge}$. Here $T^\blacklozenge$ is the right inverse of T given by the formula in Input 41. Since $TT^\blacklozenge = 1$ by construction of $T^\blacklozenge$, Lemma 39 yields $\underline{TT^\blacklozenge} = \underline{1}$. Moreover we have $\underline{TP} = \underline{0}$, because $\underline{P}$ projects onto the nullspace of $\underline{T}$. Therefore we obtain $\underline{TG} = \underline{1}$, which is indeed the fourth Moore-Penrose equation for our problem.

The problem is now reduced to finding the operator $\underline{G}$ uniquely characterized by $\underline{GT} = \underline{1 - P}$. Since $\underline{TT^\blacklozenge} = \underline{1}$, postmultiplying by $\underline{T^\blacklozenge}$ implies $\underline{G} = \underline{(1 - P)\,T^\blacklozenge}$. Hence we may choose $G = (1 - P)\,T^\blacklozenge$, and the interpretation $\underline{G}$ will be the desired generalized inverse. For any $f \in C^\infty[a, b]$, the image $u = \underline{G}\,f$ fulfills the given differential equation $\underline{T}\,u = 0$ because of the fourth Moore-Penrose equation. By the analog of Proposition 2, the range of $\underline{G}$ is the counter-image of $\underline{P}$, which fulfills the boundary conditions by the construction of P. Therefore $\underline{G}\,f$ fulfills the given BVP for any $f \in C^\infty[a, b]$, and $\underline{G}$ must coincide with the desired Green's operator due to the regularity assumption. Since $G \overset{*}{\to} G'$ in the sense of the reduction system of Input 14, Lemma 39 implies that $\underline{G} = \underline{G'}$, so G' does indeed represent the *Green's operator* as claimed.

The only claim left to prove is that G' also represents the *Green's function* in the sense explained after Equation 26. But this follows from Theorems 17 and 19 in conjunction with Definition 18: It is clear that $G = (1 - P)\,T^\blacklozenge$ cannot contain any occurrence of D, because the monomials of $1 - P$ have at most $n - 1$ occurrences of D at their end while $T^\blacklozenge$ is one monomial containing n occurrences of A. Using rules "DM" and "DA" of Input 14, the reduction will eventually in each monomial eliminate all occurrences of D and leave one occurrence of A. Hence G' consists only of monomials having the form $\mathcal{A}\mathcal{I}\mathcal{A}$

in the language of Definition 18. Now this can readily be translated into the corresponding Green's function as in Equation 27. Iterating over all monomials, we add $g_{x\leftarrow\xi}\, f$ to the first branch for a monomial $\lceil f \rceil A \lceil g \rceil$ and $g_{x\leftarrow\xi}\, f$ to the second branch for a monomial $\lceil f \rceil B \lceil g \rceil$. Monomials of the form $\mathcal{I} \lceil g \rceil$ are of course treated like $\lceil 1 \rceil \mathcal{I} \lceil g \rceil$, and those of the form $\lceil f \rceil \mathcal{I}$ like $\lceil f \rceil \mathcal{I} \lceil 1 \rceil$.

□

2 A User's Manual for the Green's Package

In the previous chapter we have presented the mathematical background of our new approach to solving regular BVPs for linear differential operators with constant coefficients. In the course of our PhD work, we have also implemented this method in the frame of the *Theorema* system, providing a collection of useful solving/computing tools that we have named the *Green's suite*. Everything is integrated smoothly into the general environment of *Theorema*, and the end user may solve a given BVP in a single call (see Section 5 of Chapter 1). In the present chapter, we want to present this suite of packages from the user's point of view; for implementation details, see the Chapter 3.

The *structure* of this chapter follows that of the Green's suite itself: In Section 1, we start out by a gentle introduction to the most important features of *Theorema*, as far as they are relevant to our present purposes. The discussion of the actual Green's suite is started in Section 2, where we present the overall architecture of the system. The remaining sections describe the three main components of the Green's suite: the polynomial reductor in Section 3, the matrix evaluator in Section 4, and finally the Green's evaluator in Section 5.

2.1 The *Theorema* Environment

As mentioned before, we have based our implementation on *Theorema,* which is designed a general-purpose tool for the working mathematician—supporting especially all tasks of proving, solving and simplifying. For a detailed description of the philosophy and capabilities of *Theorema*, we refer to [19] and [69]. For our present purposes, it will be sufficient to highlight only those features of the system that are relevant for a user who wants to solve BVPs by the Green's suite.

First of all, the user has to start up *Mathematica*, the (current) platform running the *Theorema* environment and the Green's suite. The latter two are then invoked through the following calls:

```
Needs["Theorema`"]
Needs["Theorema`Evaluators`UserEvaluators`GreenEvaluator`"]
```

As soon as loading is finished, the user will notice that the usual prompt "In[2]:=" is replaced by "TS_In[2]:=", where "TS" stands for "Theorema Standard-Session". This means that all input is interpreted as in plain *Mathematica*, only that certain additional commands are available. In particular, there are three high-level commands "Prove", "Compute" and "Solve" for assisting in the three main activities of mathematics. (In fact, the last of these is not yet supported in the current version of *Theorema*.)

For example (this material is taken from the online documentation of the **NNEqIndProver** of *Theorema*), if one wants to prove the exponential law for natural numbers, one must first specify this fact as a *Theorema* formula. This is done by entering an expression of the form **key**[**label**, any[x_1, …, x_n], **formula**], where **key** is an environment keyword, **label** a string used for referencing the formula, the x_1, …, x_n are free variables, and finally **formula** is of course the actual mathematical statement written in a very natural version of predicate logic. In our case, we would say:

Proposition[*"Add Exponents"*, any[p, n, m],
$m^{n+p} = m^n\, m^p$]

Here we have used "**Proposition**" as the environment keyword. This corresponds to a common practice in mathematical textbooks: The running text of informal explanations is typically interrupted by some formal text written as "**Proposition**: …" or similarly. Other possibilities for the environment keyword can be found out as follows:

```
$TmaEnvironmentPatterns

Definition | Definitions | Theorem | Theorems | Lemma | Lemmata | Axiom | Axioms | Corollary |
  Corollaries | Proposition | Propositions | Theory | Theories | KnowledgeBase | KnowledgeBases |
  Algorithm | Algorithms | Assumption | Assumptions | Formula | Formulae | System | Systems
```

There is *no logical difference between these keywords*; they are only provided for reasons of style. Note that, on this level, not even "**Axiom**" is logically distinguished from "**Theorem**" and analogous names for mathematical statements. The reason is that a formula may well serve as a definition at one time although it is proved as a normal mathematical theorem at some other time; the axioms of the real numbers are an example, because they may be proved if one selects a specific construction like Dedekind cuts for them. Similar remarks hold true for the environment "**Definition**". The actual role of a formula is specified in the proof call—whether it is to be proved or to be used as an assumption.

The above proposition has the *label "Add Exponents"*. Just as the environment label, there is no logical significance to such labels. They are merely used for referring to formulae. For example, in a proof, a certain step may be justified by saying: Using (**Proposition** (*Add Exponents)*), this implies… In fact, the environment keyword and the environment label may be thought of as making up a compound label; in our example, this is just (**Proposition** (*Add Exponents)*).

The *free variables* are p, n, m in the sample proposition considered above. Logically, they are universally quantified (this corresponds to the common practice in logic of taking the universal closure of formulae with free variables). Hence one may specify the same mathematical statement by the following equivalent characterization:

Proposition[*"Add Exponents"*,

$$\forall_{p,n,m} \; m^{n+p} = m^n \, m^p \,]$$

Here we have *no free variables*, but the formula contains a corresponding universal quantifier. So the difference between these two versions is again just a question of style, similar to mathematical textbooks. One may either say

(1) Now for any p, n, m, we have

$$m^{n+p} = m^n \, m^p .$$

Or one says the following:

(2) Now we have

$$\forall_{p,n,m} \; m^{n+p} = m^n \, m^p .$$

These issues are irrelevant on the logical level of theorems, but they are crucial for their *natural presentation*. For example, it is usually considered better style to avoid universal quantifiers when their scope is the whole formula, thus preferring version (1) over version (2) above.

Having specified the proof goal, one must also provide the relevant knowledge base to be used in the proof. Obviously, this must include the definition of exponentiation itself. In our case, it is this:

Definition[*"Exponentiation"*, any[m, n],
$m^0 := 0^+$ "exp 0"
$m^{n^+} := m^n\, m$ "exp ."]

Note that we have added the *formula labels* here, namely *"exp 0"* and *"exp ."* in addition to the overall environment label *"Exponentiation"*. This is useful for referring to the first or second clause on an individual basis, thus giving rise to a hierarchic compound label consisting of three components—the environment keyword, the environment label, the formula label. For example, the formula $m^0 := 0^+$ would be referenced by (**Definition** (*Exponentiation: exp 0)*).

The usage of the ":=" sign instead of the normal "=" sign is a message on the meta level, signifying *definitional equalities*. This can be used as a hint for some provers, because definitions—unlike other equalities—are often used only from left to right. From the logical viewpoint, $a = b$ is of course the same as $a := b$.

Now one could add the definitions of addition and multiplication (which are used in exponentiation) in an analogous manner. In *Theorema*, however, we consider it more appropriate to build up mathematical theories in a layered approach; see [18] for a detailed account. Hence we prove some crucial properties of addition and multiplication beforehand, and we collect them all in an appropriate knowledge base:

Theory[*"Properties of +, ∗"*,

Definition[*"Multiplication"*]
Proposition[*"Multiplication from Left"*]
Proposition[*"Left Distributivity"*]
Proposition[*"Right Distributivity"*]
Proposition[*"Multiplication of One from Left"*]
Proposition[*"Multiplication of One from Right"*]
Proposition[*"Multiplication of Zero from Left"*]
Theory[*"Addition"*]]

The construct **Theory** is used here for aggregating formulae into a common pool. It contains the *definition of multiplication*:

Definition[*"Multiplication"*, any[m, n],
$m * 0 = 0$ "∗0"
$m * n^+ = (m * n) + m$ "∗ ^+"]

Then we have included *six crucial properties of multiplication*, contained in the following propositions:

Proposition[*"Multiplication from Left"*, any[m, n],
$m^+ * n = (m * n) + n$]
Proposition[*"Left Distributivity"*, any[m, n, p],
$m * (n + p) = m * n + m * p$]

Proposition[*"Right Distributivity"*, any[m, n, p],
$(m + n) * p = m * p + n * p$]
Proposition[*"Multiplication of One from Left"*, any[m],
$0^+ * m = m$]
Proposition[*"Multiplication of One from Right"*, any[m],
$m * 0^+ = m$]
Proposition[*"Multiplication of Zero from Left"*, any[n],
$0 * n = 0$]

Finally, we have added the whole theory of addition, which comes from an even earlier "exploration layer". It is again made up of a definition and two propositions, thus demonstrating the nesting *capability* of the **Theory** construct:

Theory["Addition",
Definition[*"Addition"*]
Proposition[*"Addition of Zero from Left"*]]
Proposition[*"Addition from Left"*]

This is the *definition of addition*:

Definition["Addition", any[m, n],
$m + 0 = m$ $\quad$ *" +0"*
$m + n^+ = (m + n)^+$ $\quad$ *" + succ"*]

And here are the *two propositions* added to it for making up the theory of addition:

Proposition[*"Addition of Zero from Left"*, any[n],
$0 + n = n$ "0 + "]
Proposition[*"Addition from Left"*, any[m, n],
$m^+ + n = (m + n)^+$]

Besides this knowledge, there are four additional properties that are so crucial that one wants to treat them in a special way: associativity and commutativity of both addition and multiplication. The reason is that it would be extremely tedious to mention each use of these properties explicitly (although it is of course possible), and using them without explicit reference allows a highly efficient implementation via controlled delegation to *Mathematica*. We can specify such a property of an operation $\circ$ in *Theorema* by the following item:

Property[$\circ \to$ {Associative, Commutative}]

Of course, one may leave out either of the properties "Associative" or "Commutative" if this is desired. The important point is that one has to add these items to the implicit knowledge base, which is specified via the option **built-in** of the **Prove** command. All the other knowledge gathered above would of course go into the explicit knowledge base, specified by the corresponding option **using**. Hence we will have to use the options

using $\to$ {**Definition**[*"Exponentiation"*], **Theory**[*"Properties of +, *"*]}

and

built-in $\to$ {**Property**[+ $\to$ {Associative, Commutative}], **Property**[$*$ $\to$ {Associative, Commutative}]}

for specifying all the relevant knowledge.

Now we have specified all the logical ingredients for the proof task: the goal formula and all the assumptions that may be used. The next thing to fix is the *special prover* to be applied. This leads us to another major design feature of *Theorema*. Unlike other proving communities, we do

not believe in a monolithic prover that takes care of the whole of mathematics. In our opinion, such a prover would not be a reasonable goal: Just compare the notion of such a universal prover to a universal solver for all of mathematics—taking care of linear equation systems, partial differential equations, combinatory logic, … at the same time. Nobody would consider it realistic to design such a universal solver. So why should one aspire to have an analogous thing for proving, which is even more complex than solving? (Note that one can regard solving as the special case of constructively proving formulae having the form $\exists_x S_x = T_x$, where x is one or more variables and T_x, S_x are terms normally containing some of the variables x.)

Hence there is no such thing as "the *Theorema* prover". Every mathematical domain comes with its own natural provers that are special to this domain (which is usually characterized through a set of axioms). Of course, one may also consider the absolutely general domain (characterized by the empty set of axioms), which is covered in *Theorema* by the **PredicateProver**. However, we do not give particular preference to any domain of mathematics, thus putting the predicate prover on an equal basis with, say, the *prover for natural numbers* called **NNEqIndProver**. And the latter is exactly what we need for our problem, hence we have to say **by → NNEqIndProver**, where **by** is the option used for selecting the special prover. The crucial point about this prover is of course that it assumes the inductive domain of natural numbers—thus presupposing the corresponding induction axioms, which are used in the from of suitable inference rules.

Each special prover accepts a certain number of *options* for controlling its behavior; they are specified by the option **ProverOptions** given to the **Prove** command. For seeing which options are supported by **NNEqIndProver** and which default values are used, we can issue the following call:

```
Options[NNEqIndProver]

{ConstOrder → {{SuperPlus, +, ∗}}, NNRepr → Nondecimal,
  SpecialSimplification → True, TermOrder → RPLexOrder}
```

Here we can see that the *order of the constant symbols* can be specified, and the setting is already close to what we need: In the goal formula and the assumptions, the only function symbols besides = and := is SuperPlus (the successor function, signified by a superscript plus symbol, e.g. in m^+), +, ∗, and Power (the exponentiation function, signified by all other superscripts, e.g. in m^n). Hence we will simply append the Power function to the default list given above.

The *other options* may be left on their default values: The option **NNRepr** controls the representation of natural numbers; for example, 3 is represented as 0^{+++} with the default setting "Nondecimal" and as $0 + 1 + 1 + 1$ if the setting is changed to "Decimal". Activating the option **SpecialSimplification** allows the prover to contract several simplifications into one step, thus making the proof much more compact; the option is activated by default. Finally, the option **TermOrder** is a kind of strategy used in equational proving, which usually need not be changed from its default value.

Actually, all this information is also available online if one asks for it in the usual *Mathematica* way:

```
?? ConstOrder
☑ Option that specifies the order of the constants in the rewrite process.
?? NNRepr
Option for the NIP prover, to be used in a Prove call, that specifies which
☑    inductive representation to use for natural numbers. If 'Nondecimal',  they are
     constructed from 0 and SuperPlus, if 'Decimal' they are constructed from 0 and +1.
?? SpecialSimplification
☑ Option for simplification: if True, then it performs
     in one step simplification with respect to built−in knowledge.
```

Finally, we will provide some specification about the proof presentation. By default, the prover shows everything that it did, including also failed proof attempts. This may be very interesting for intermediate analysis, but usually one wants to see a polished proof in the end. In *Theorema*, this can be accomplished by using *proof simplification.* The corresponding option is **transformBy**, which specifies the proof transformer to be used as a postprocessor to the prover. Besides proof simplification, this could also be e.g. a failure analyzer or a conjecture generator. In our case, however, all we want to have is plain proof simplification, so we use the standard transformer for this purpose, called **ProofSimplifier**. It accepts some options controlling the kind of simplification one desires. For seeing the possibilities, we can say this:

```
Options[ProofSimplifier]

{branches → All, steps → All, substitutions → All}
```

The option **branches** specifies which branches of the proof tree one wants to see. The default setting is to show everything, including the failed branches. We want to see *only the proved ones*, which can be accomplished by setting this option to the value "Proved" (one can also set it to "Disproved" if the prover is used for refuting formulae, and one can even set it to "Failed" if one wants to analyze just the failed proof attempts). The option **steps** could be employed for making the proof output more compact by condensing certain combinations of proof steps (see the *Theorema* documentation for details, e.g. by saying "?? Essential", "?? Combined", "?? Useful", "?? Lifted", "?? LiftedParallel" as to be expected from the output below); we will not make use this possibility. Finally, the option **substitutions** can be used for restricting the substitutions generated by certain provers to the "Useful" ones; this is not needed in our case. Again, one can get all the essential information in the following online documentation:

```
?? branches
☑ Option of ProofSimplifier with possible values: Proved, Pending,
    Failed, Disproved and list combinations of these. All (default) means list of all.
?? steps
☑ Option of ProofSimplifier with possible values: All, Essential,
    Combined, Useful, Lifted, LiftedParallel and list combinations of these.
?? substitutions
☑ Option of ProofSimplifier with possible values: All, Useful.
```

Now we have all the relevant data for issuing the *prove call* necessary in our example:

```
Prove[Proposition["Add Exponents"],
  using → {Definition["Exponentiation"], Theory["Properties of +, *"]},
  built-in → {Property[+ → {Associative, Commutative}], Property[* → {Associative, Commutative}]},
  by → NNEqIndProver, ProverOptions → {ConstOrder → {{SuperPlus, +, *, Power}}},
  transformBy → ProofSimplifier,  TransformerOptions → {branches → Proved}]

- ProofObject -
```

What we get is the following *induction proof* of the goal formula (note that everything is produced completely automatically up to the □ symbol!):

We prove (Proposition (Add Exponents)) by induction on p.

Induction Base:

(1) $\forall_{n,m} (m^{n+0} = m^n\, m^0)$.

We take in (1) all variables arbitrary but fixed and prove:

(4) $m_1^{n_1+0} = m_1^{n_1}\ m_1^{0}$.

A proof by simplification of (4) works.

Simplification of the lhs term:

$m_1^{n_1+0}$ = by (Definition (Addition): +0)

$m_1^{n_1}$ ⌋

Simplification of the rhs term:

$m_1^{n_1}\ m_1^{0}$ = by (Definition (Exponentiation): exp 0)

$m_1^{n_1}\ 0^{+}$ = by (Definition (Multiplication): * ^+)

$m_1^{n_1}\ 0 + m_1^{n_1}$ = by (Definition (Multiplication): *0)

$0 + m_1^{n_1}$ = by (Special Simpl)

$m_1^{n_1} + 0$ = by (Definition (Addition): +0)

$m_1^{n_1}$ ⌋

Induction Step:

Induction Hypothesis:

(2) $\underset{n,m}{\forall}\ (m^{n+p_1} = m^n\ m^{p_1})$

Induction Conclusion:

(3) $\underset{n,m}{\forall}\ (m^{n+p_1^{+}} = m^n\ m^{p_1^{+}})$.

We take in (3) all variables arbitrary but fixed and prove:

(5) $m_2^{n_2+p_1^{+}} = m_2^{n_2}\ m_2^{p_1^{+}}$.

A proof by simplification of (5) works.

Simplification of the lhs term:

$m_2^{n_2+p_1^{+}}$ = by (Definition (Addition): + succ)

$m_2^{(n_2+p_1)^{+}}$ = by (Definition (Exponentiation): exp .)

$m_2^{n_2+p_1}\ m_2$ = by (2)

$(m_2^{n_2}\ m_2^{p_1})\ m_2$ = by (Special Simpl)

$m_2\ (m_2^{n_2}\ m_2^{p_1})$ ⌋

Simplification of the rhs term:

$m_2^{n_2}\ m_2^{p_1^{+}}$ = by (Definition (Exponentiation): exp .)

$m_2^{n_2}\ (m_2^{p_1}\ m_2)$ = by (Special Simpl)

$m_2\ (m_2^{n_2}\ m_2^{p_1})$ ⌋

□

Of course, the prover did *not use all the available knowledge*; in fact, we have only used the definitions and some special simplification. This is very similar to the situation of human proving: In a typical situation, we do not know in advance which formulae will ultimately turn out to be necessary. (Of course, we could make the work of the prover easier by preselecting those formulae that will actually be used for the proof.)

Using the proposition just proved, we could now go on proving another proposition about exponentiation, thus continuing this exploration layer. For example, it is very natural to ask about the products of powers with the same exponents:

Proposition["*Multiply Terms with Same Exponents*", any[n, p, m],
  $m^n * p^n = (m * p)^n$]
Prove[**Proposition**["*Multiply Terms with Same Exponents*"],
  using → {**Proposition**["*Add Exponents*"], **Definition**["*Exponentiation*"], **Theory**["*Properties of* +, ∗"]},
  built-in → {**Property**[+ → {Associative, Commutative}], **Property**[∗ → {Associative, Commutative}]},
  by → NNEqIndProver, ProverOptions → {ConstOrder → {{SuperPlus, +, ∗, Power}}},
  transformBy → ProofSimplifier, TransformerOptions → {branches → Proved}]

- ProofObject -

This produces a fairly *similar induction proof*, again using only the definitions (in particular, not using the proposition proved just before).

We prove (Proposition (Multiply Terms with Same Exponents)) by induction on n.

Induction Base:

(1) $\forall_{p,m} (m^0\ p^0 = (m\ p)^0)$.

We take in (1) all variables arbitrary but fixed and prove:

(4) $m_1{}^0\ p_1{}^0 = (m_1\ p_1)^0$.

A proof by simplification of (4) works.

Simplification of the lhs term:

$m_1{}^0\ p_1{}^0 =$ by (Definition (Exponentiation): exp 0)

$0^+\ p_1{}^0 =$ by (Definition (Exponentiation): exp 0)

$0^+\ 0^+ =$ by (Definition (Multiplication): * ^+)

$0^+\ 0 + 0^+ =$ by (Definition (Multiplication): *0)

$0 + 0^+ =$ by (Special Simpl)

$0^+ + 0 =$ by (Definition (Addition): +0)

0^+ ⌋

Simplification of the rhs term:

$(m_1\ p_1)^0 =$ by (Definition (Exponentiation): exp 0)

0^+ ⌋

Induction Step:

Induction Hypothesis:

(2) $\forall_{p,m} (m^{n_1}\ p^{n_1} = (m\ p)^{n_1})$

Induction Conclusion:

(3) $\forall_{p,m} \left(m^{n_1{}^+}\ p^{n_1{}^+} = (m\ p)^{n_1{}^+}\right)$.

We take in (3) all variables arbitrary but fixed and prove:

(5) $m_2{}^{n_1{}^+}\ p_2{}^{n_1{}^+} = (m_2\ p_2)^{n_1{}^+}$.

A proof by simplification of (5) works.

Simplification of the lhs term:

$m_2{}^{n_1{}^+}\ p_2{}^{n_1{}^+}$ = by (Definition (Exponentiation): exp .)

$(m_2{}^{n_1}\ m_2)\ p_2{}^{n_1{}^+}$ = by (Definition (Exponentiation): exp .)

$(m_2{}^{n_1}\ m_2)\ (p_2{}^{n_1}\ p_2)$ = by (Special Simpl)

$m_2\ (p_2\ (m_2{}^{n_1}\ p_2{}^{n_1}))\rfloor$

Simplification of the rhs term:

$(m_2\ p_2)^{n_1{}^+}$ = by (Definition (Exponentiation): exp .)

$(m_2\ p_2)^{n_1}\ (m_2\ p_2)$ = by (2)

$(m_2{}^{n_1}\ p_2{}^{n_1})\ (m_2\ p_2)$ = by (Special Simpl)

$m_2\ (p_2\ (m_2{}^{n_1}\ p_2{}^{n_1}))\rfloor$

□

Finally, let us do one more proof in this exploration layer of exponentation.

```
Proposition["Multiply Exponents", any[n, p, m],
      m^(n*p) = (m^n)^p ]
Prove[Proposition["Multiply Exponents"],
  using → {
      Proposition["Add Exponents"],
      Proposition["Multiply Terms with Same Exponents"],
      Definition["Exponentiation"],
      Theory["Properties of +, *"]},
  by → NNEqIndProver, ProverOptions → {ConstOrder → {{SuperPlus, +, *, Power}}},
  transformBy → ProofSimplifier, TransformerOptions → {branches → Proved}]
```

▪ ProofObject ▪

This produces the following induction proof:

We prove (Proposition (Multiply Exponents)) by induction on n.

Induction Base:

(1) $\forall_{p,m} (m^{0\,p} = (m^0)^p)$.

We take in (1) all variables arbitrary but fixed:

(4) $m_1{}^{0\,p_1} = (m_1{}^0)^{p_1}$

and simplify it.

Simplification of the lhs term:

$m_1{}^{0\,p_1}$ = by (Proposition (Multiplication of Zero from Left))

$m_1{}^{0}$ = by (Definition (Exponentiation): exp 0)

0^{+} ⌋

Simplification of the rhs term:

$(m_1{}^{0})^{p_1}$ = by (Definition (Exponentiation): exp 0)

$(0^{+})^{p_1}$ ⌋

Hence, it is sufficient to prove:

(5) $\underset{p}{\forall}\,(0^{+} = (0^{+})^{p})$.

We prove (5) by induction on p.

Induction Base:

(6) $0^{+} = (0^{+})^{0}$.

A proof by simplification of (6) works.

Simplification of the lhs term:

0^{+} ⌋

Simplification of the rhs term:

$(0^{+})^{0}$ = by (Definition (Exponentiation): exp 0)

0^{+} ⌋

Induction Step:

Induction Hypothesis:

(7) $0^{+} = (0^{+})^{p_2}$

Induction Conclusion:

(8) $0^{+} = (0^{+})^{p_2{}^{+}}$.

A proof by simplification of (8) works.

Simplification of the lhs term:

0^{+} ⌋

Simplification of the rhs term:

$(0^{+})^{p_2{}^{+}}$ = by (Definition (Exponentiation): exp .)

$(0^{+})^{p_2}\ 0^{+}$ = by (7)

$0^{+}\ 0^{+}$ = by (Definition (Multiplication): * ^+)

$0^+\ 0 + 0^+ \ =$ by (Definition (Multiplication): *0)

$0 + 0^+ \ =$ by (Definition (Addition): + succ)

$(0 + 0)^+ \ =$ by (Definition (Addition): +0)

0^+ ⌋

Induction Step:

Induction Hypothesis:

(2) $\underset{p,m}{\forall}\ (m^{n_1\ p} = (m^{n_1})^p)$

Induction Conclusion:

(3) $\underset{p,m}{\forall}\ \left(m^{n_1{}^+\ p} = (m^{n_1{}^+})^p\right)$.

We take in (3) all variables arbitrary but fixed and prove:

(9) $m_2{}^{n_1{}^+\ p_3} = (m_2{}^{n_1{}^+})^{p_3}$.

A proof by simplification of (9) works.

Simplification of the lhs term:

$m_2{}^{n_1{}^+\ p_3} \ =$ by (Proposition (Multiplication from Left))

$m_2{}^{n_1\ p_3 + p_3} \ =$ by (Proposition (Add Exponents))

$m_2{}^{n_1\ p_3}\ m_2{}^{p_3}$ ⌋

Simplification of the rhs term:

$(m_2{}^{n_1{}^+})^{p_3} \ =$ by (Definition (Exponentiation): exp .)

$(m_2{}^{n_1}\ m_2)^{p_3} \ =$ by (Proposition (Multiply Terms with Same Exponents))

$(m_2{}^{n_1})^{p_3}\ m_2{}^{p_3} \ =$ by (2)

$m_2{}^{n_1\ p_3}\ m_2{}^{p_3}$ ⌋

□

This time we used some *explicit non-definitional knowledge*: The proposition about multiplying with zero from the left, and the proposition about the product of powers with the same exponent proved in the beginning. Note, however, that we have not put any a-priori information about this to the prover—it got the full knowledge base just as in the other two prove calls, containing numerous other assumptions that turned out to be unnecessary for this particular proof.

Analyzing the internal setup of the prover **NNEqIndProver**, we can see that it actually could use three different "types of reasoning":

- Some custom-tailored inference rules for the special predicate =, called *rewrite rules*. In particular, the proof of a formula $S = T$ is accomplished by transforming both S and T to their normal form and then checking whether these coincide. The transformations to normal form are announced by the words "Simplification of the lhs/rhs term" in the above proofs. They make implicit use of the symmetry and replacement axioms for equality.

- The usual *inference rules of predicate logic*, which are valid in any domain, not only in the realm of natural numbers. Hence the above-mentioned universal proof engine **Predicate-Prover** consists of only this basic prover. These rules were unnecessary in the above proofs. In fact, we would have to use the prover **NNIndProver** if we needed them as well.

- Finally, the decisive inference rule used in all three proofs above is *induction*: For proving a goal of the form $\forall_n \Phi_n$, where Φ_n is a formula typically containing a free occurrence of the variable n, it suffices to prove $\Phi_{n\leftarrow 0}$ and to prove $\Phi_{n\leftarrow m^+}$ under the assumption of $\Phi_{n\leftarrow m}$, where m is an arbitrary but fixed natural number.

In order to make things a bit more general, let us now analyze the internal structure of the prover **NNIndProver** rather than the prover **NNEqIndProver** used for the proofs above. As just explained, it is composed of three sets of reasoning rules—rewrite logic, predicate logic, induction. In *Theorema*, this is realized by composing the so-called user prover **NNIndProver** from the three corresponding blocks, called *basic provers*. The basic prover of rewrite logic is called **Simplifier**, the natural-numbers induction prover is called **NIP**, and the predicate-logic prover using a kind of natural deduction calculus is called **PND**. This could be seen in the implementation of the user prover **NNIndProver**, which contains the following crucial command:

```
AddConstraints[•nonExclusive[True, Simplifier],
  •nonExclusive[True, NIP],
  •nonExclusive[True, PND]]
```

This command joins the three basic provers to make up the corresponding user prover. However, we will not enter into any implementation issues here, since we just want to provide a rough overview of the general *Theorema* environment. We have only mentioned it here, because we will encounter an analogous structure in the Green's evaluator to be described in the next section.

In fact, recent discussions within the *Theorema* group drive towards a prover setup that will allow an *even greater level of flexibility*, changing between proving / solving / computing situations on a per-rule basis. As these discussions are still under way and the current *Theorema* system has the setup of user provers and basic provers as described above, we have built the Green's suite along these lines of the user reasoner / basic reasoner setup. (The notion of reasoning subsumes proving, solving, computing—the three fundamental activities of mathematics.)

2.2 The Overall Design of the Green's Suite

We have organized the Green's suite as a *one user evaluator* named **GreenEvaluator** appealing to *three basic evaluators* named **ReduceNoncommutativePolynomial**, **EvaluateMatrices**, **EvaluateStandard**—just like the user prover **NNIndProver** is made up of the three basic provers **Simplifier**, **NIP**, **PND**. Let us briefly describe the three basic evaluators involved in the Green's suite:

- The basic evaluator **ReduceNoncommutativePolynomial** is used for *reducing a noncommutative polynomial* to its normal form with respect to a given system of noncommutative polynomial equalities. For example, reducing $A\,D - D\,A$ with respect to the equalities $A\,D = 1$, $D\,A = 1 - L$ would yield the result $2 - L$.

- The most important *matrix operations* like addition, multiplication and inversion are carried out by the basic evaluator **EvaluateMatrices**.

- Unlike the other two basic evaluators, **EvaluateStandard** is already provided by the standard *Theorema* environment (hence one can find its description in the online documentation so that we will not describe it here). Its main purpose is *unfolding definitions*. For example, it will unfold the term $1 + D^\blacklozenge$ into $1 + A$, using the definition of $(\ldots)^\blacklozenge$ as right inversion; see Input 41 of Chapter 1.

We will describe the two basic evaluators **ReduceNoncommutativePolynomial** and **EvaluateMatrices** in the next two sections from a generic point of view, meaning that we will not yet consider the particular purpose we have in mind when using them for solving BVPs. In fact, the idea of basic evaluator is to be an *independent unit of reasoning* that provides custom-tailored evaluation for certain specific domains—in this case, polynomials and matrices (whereas the evaluator **EvaluateStandard** works for the general domain just like the predicate prover **PND** for proof tasks in the general domain).

Having these general components (as well as many other basic provers and evaluators and solvers), one can imagine constructing a lot of *custom-tailored methods* by specializing them to the particular problem one has in mind—each specialization corresponding to a user provers, a user evaluator or a user solver. The **GreenEvaluator** is but one of them, and we will describe its specific structure in the last section.

The **GreenEvaluator** is the first example of such a user evaluator, constructed in parallel to the notion of user provers. Hence it will not come as a surprise that there a number of issues that have *not yet been straightened out completely*. Let us mention the most important ones.

What proof objects are to proving, *trace objects* are to computing: A record containing all the relevant information about how and why a certain step was done. But whereas there is a clear and stabilized structure for proof objects in *Theorema*, this is not yet the case for trace objects. For example, there is no meta-evaluator that would take care of assembling trace objects from various basic evaluators just as the meta-prover does for proof objects coming from various basic provers. While we do provide tracing for the basic evaluator **ReduceNoncommutativePolynomial** (which will be described below) and the standard *Theorema* system does it for the basic evaluator **EvaluateStandard**, we do not yet have tracing support for the basic evaluator **EvaluateMatrices**. Hence there is also no trace for the overall computing procedure effected by the user evaluator **GreenEvaluator**.

For practical purposes, it is very important to have an *intuitive and natural notation* for the key concepts used. As this is also an important guideline for the *Theorema* framework in general, we have tried to pay close attention to this issue. The reader may convince herself in the subsequent sections that she does not have to use ugly-looking or even ASCII syntax for input and output. We have accomplished this via the usual mechanism of **MakeExpression** and **MakeBoxes** as it is provided by *Mathematica* and extensively used in the whole of *Theorema*. However, we have not yet constructed a uniform and user-transparent framework for handling notation in *Theorema*. This means that as soon as a certain prover or evaluator is loaded, one "buys" all the notation it provides, and there is no way to undo this (except reloading *Theorema* of course). For example, the notation $\hat{a} + \hat{b}$ denotes matrix addition as soon as **EvaluateMatrices** is loaded.

In relation with the basic evaluator **ReduceNoncommutativePolynomial**, there is also a relevant prover for establishing the *confluence* of certain polynomial rewrite systems. We have also implemented some ad-hoc functions doing this job, and we have used this in Computation 29 of Chapter 1 (reducing 233 S-polynomials by hand would really not be fun at all). However, this prover is not yet integrated in the usual framework of *Theorema* provers, so we will not describe it in detail here. We will rather regard it as a convenient "tool for proving confluence", and as such we will describe it briefly in Section 3.

The user evaluator named **GreenEvaluator** is actually a prime example of where we would like to apply the flexible concept of *rule-based integration for proving / solving / computing situations* briefly mentioned at the close of Section 1. As stated at the outset of this thesis, our main concern is to solve(!) BVPs. But still our main tool for doing this is the evaluator(!) named **GreenEvaluator**. The reason for this is of course that we reduce the problem of solving(!) a certain equation for a function to computing(!) a related operator for this problem (here "computing" and "evaluating" may be regarded as synonymous—see the discussion at the beginning of Section 3); see the motivation given in Section 1 of Chapter 1. Still it would be very natural to add one more reasoning rule that expresses just this transition of the solve situation to the corresponding compute situation such that one could issue a call like

$$\begin{aligned}&\text{Solve}[u'' = f \wedge u[0] = 0 \wedge u[1] = 0,\\ &\quad \text{for}[u],\ \text{given}[f],\ \text{by} \to \text{GreenSolver}]\end{aligned}$$

rather than the corresponding call

$$\begin{aligned}&\text{Compute}[\text{Green}[D^2, \langle L, R\rangle],\\ &\quad \text{by} \to \text{GreenEvaluator}]\end{aligned}$$

to be used now (see the Section 5 for a detailed description of what the call above means).

Finally, let us make one linguistic remark: We have called the basic evaluator for noncommutative polynomials by the name **ReduceNoncommutativePolynomial** rather than **EvaluateNoncommutativePolynomial** for the following reason: Although we could identify the terms "reduce", "evaluate", "simplify", "transform", "normalize", and "compute" as synonymous on logical grounds (meaning nothing else than applying rewrite rules until they are saturated), one can observe certain *usage distinctions* in common language:

- For polynomials, *evaluation* is already preoccupied for the evaluation homomorphism carrying e.g. $\text{eval}[x^2 + 3, 2]$ to 7; this is why we have avoided it in the name for our basic evaluator. In a more general sense, evaluation is usually associated with unfolding definitions. Therefore it is an appropriate name for **EvaluateStandard** as well as **EvaluateMatrices**.

- The term *computing* is the most general in our opinion. Some people would restrict it to ground terms, because this is a very typical usage of this word, e.g. when one says that computing $3 + 7$ gives 10. However, we think that it is also often applied in a more comprehensive sense, e.g. in the phrases "symbolic computation" or "computing all the solutions of a given differential equation".

- If one uses the term *simplifying,* the emphasis is obviously on making the given term simpler. This may be so with respect to some (maybe implicit) term ordering or in the sense of canonicality (a canonical form is in a sense always the simplest form possible). Hence it does not really cover the more general case of applying an arbitrary system of rewrite rules, since such a system may neither be orientable with respect to some term ordering nor admit canonical forms.

- The term *transformation* denotes a very general concept that may apply to all of reasoning, including transitions between proving / solving / computing situations. However, we will use this term below only for a slight generalization that subsumes what we call normalization and reduction of polynomials.

- Somewhat more precise, the term *normalizing* refers explicitly to obtaining the normal form (which may or may not be the canonical form, depending on whether the rewrite

system is confluent). In principle, we could use this term, because the results we obtain are indeed normal forms of the rewrite system induced by the polynomial axioms and the given polynomial equalities. However, this could be misleading, because normalizing a polynomial usually refers to the much narrower process of just expanding the polynomial according to the polynomial axioms.

- Finally, there is the term *reducing*. Although it does suggest decreasing some(!) measure similar to the term "simplifying", where this measure is obviously the complexity (thus increasing the simplicity), its scope is usually much wider: Any(!) application of rewrite rules can be considered as a reduction, namely regarding the number of redexes. This is in fact the usual term (more precisely, it is β-reduction) used in lambda calculus, which is in a sense the general theory of rewriting. Hence we can also apply it to the rewrite system that is generated by the given polynomial equalities—based on the usual polynomial axioms. (Note that we have avoided the term "rewriting" itself because it is too technical, although one could regard it as synonymous to "reducing".)

2.3 The Reductor for Noncommutative Polynomials

The basic evaluator **ReduceNoncommutativePolynomial** is a general-purpose tool for *reducing a noncommutative polynomial* to its normal form with respect to some given system of polynomial equalities. It operates on $R\langle X \mid X \in \Xi\rangle$, where Ξ is some set of indeterminates that must be specified and R is some ring containing $\mathbb{C}$. (Typically one deals with rings of complex functions, where the complex numbers are naturally embedded as constant functions.)

Using the approach outlined in Section 4 of the Appendix, it applies two kinds of *polynomial transformations*:

- Those following certain polynomial axioms like $X(Y+Z) \to XY + XZ$, which we will subsume under the heading *polynomial normalization.*
- Those using a certain polynomial equality from the given system like $XY = Y + Z$ for obtaining $XY + XZ \to (Y+Z) + XZ$. We will call this process *polynomial reduction.* Obviously these are the core steps for a transformation chain; the interspersed normalization steps should rather be regarded as "low-level" computations getting the data structures straight. Hence the name **ReduceNoncommutativePolynomial**.

Let us now see how this looks like in *Theorema*. Assuming the Green's suite is already loaded as explained above, we set the evaluator **ReduceNoncommutativePolynomial** as the default one so that we do not have to mention it explicitly all the time. Besides this, we must also deactivate the option **UseFlattenedDefaults**, which is activated by default; see the online help listed above. (The reason is that whereas it is usually preferable to transform equalities into Mathematica rewrite rules once and for all at the beginning of a computation, we cannot do this now because polynomial equalities have to be treated in a specialized way.)

```
SetOptions[Compute, by → ReduceNoncommutativePolynomial, UseFlattenedDefaults → False];
```

?? UseFlattenedDefaults

☑ An option for 'Compute'. 'True' means that the flattened default knowledge bases are joined to the respective lists of transformation rules that are obtained from the knowledge bases given in the options 'using→' and 'built-in→'. 'False' means that the default knowledge bases in their original structure as environments are adjoined to the given knowledge bases and are then flattened. It may lead to unexpected results if the default evaluator is changed between the moment you specify default knowledge and the moment you call 'Compute'.

We will start with some trivial examples involving only normalization in $\mathbb{C}\langle X, Y, Z\rangle$. So for the moment let us fix X, Y, Z as the *indeterminates*. And let us produce *no tracing information* as there is nothing essential to show without proper reductions. We do this by setting the corresponding options of the basic evaluator:

```
SetOptions[ReduceNoncommutativePolynomial,
    Indeterminates → {X, Y, Z}, inNotebook → "None"];
```

So here are some examples involving *only normalization*:

```
Compute[X (Y + Z)]
```

$$X\,Y + X\,Z$$

```
Compute[(X + Y) (Y + Z) (Z + X) − X Y Z]
```

$$X\,Y\,X + X\,Z^2 + X\,Z\,X + Y^2\,Z + Y^2\,X + Y\,Z^2 + Y\,Z\,X$$

```
Compute[(X − Y − 1) (X + Y − 1)]
```

$$1 - 2\,X + X^2 + X\,Y - Y\,X - Y^2$$

Note the *lack of commutation* in the previous example—in commutative polynomials, the terms $X\,Y$ and $-Y\,X$ would cancel. And this is indeed the case if we use identifiers that are not declared as indeterminates (as X, Y, Z are), because the default assumption is that any identifiers like A, B represent unknown complex numbers:

```
Compute[(A − B − 1) (A + B − 1)]
```

$$1 - 2\,A + A^2 - B^2$$

Now let us construct a small system of *polynomial equalities*:

```
System["Test Equalities",
  X Y = Y + Z                    "1"
  Y Z = Z + X                    "2"
  Z X = X + Y                    "3"
]
```

We can use this system for *reducing polynomials*:

```
Compute[X Y Z X,
  using → System["Test Equalities"]]
```

$$2\,X + 2\,Y + X^2 + Z\,Y$$

```
Compute[X^2 Y^2 Z^2,
  using → System["Test Equalities"]]
```

$$2\,X + 2\,Z + 2\,X\,Z + 2\,Z^2 + X^2\,Z + Y\,X\,Z + Z^3 + X\,Z^3$$

In this case, it could be interesting to see some *trace information*. So let us change the options again so that tracing is supported, and then let us redo the two examples of above:

```
SetOptions[ReduceNoncommutativePolynomial,
    inNotebook → "Current"];
Compute[X Y Z X,
  using → System["Test Equalities"]]
```

We compute:

$$X\,Y\,Z\,X \overset{(\ldots)}{\downarrow =}$$

$$\boxed{X\,Y}\,Z\,X \overset{(1)}{\downarrow =}$$

$$\boxed{Y\,Z}\,X + Z^2\,X \overset{(2)}{\downarrow =}$$

$$X^2 + \boxed{Z\,X} + Z^2\,X \overset{(3)}{\downarrow =}$$

$$X + Y + X^2 + Z\boxed{Z\,X} \overset{(3)}{\downarrow =}$$

$$X + Y + X^2 + \boxed{Z\,X} + Z\,Y \overset{(3)}{\downarrow =}$$

$$2\,X + 2\,Y + X^2 + Z\,Y \qquad \square$$

```
2 X + 2 Y + X^2 + Z Y
Compute[X^2 Y^2 Z^2,
  using → System["Test Equalities"]]
```

We compute:

$$X^2\,Y^2\,Z^2 \overset{(\ldots)}{\downarrow =}$$

$$X\boxed{X\,Y}\,Y\,Z^2 \overset{(1)}{\downarrow =}$$

$$\boxed{X\,Y}\,Y\,Z^2 + X\,Z\,Y\,Z^2 \overset{(1)}{\downarrow =}$$

$$Y\boxed{Y\,Z}\,Z + Z\,Y\,Z^2 + X\,Z\,Y\,Z^2 \overset{(2)}{\downarrow =}$$

$$\boxed{Y\,Z}\,Z + Y\,X\,Z + Z\,Y\,Z^2 + X\,Z\,Y\,Z^2 \overset{(2)}{\downarrow =}$$

$$X\,Z + Z^2 + Y\,X\,Z + Z\boxed{Y\,Z}\,Z + X\,Z\,Y\,Z^2 \overset{(2)}{\downarrow =}$$

$$X\,Z + Z^2 + Y\,X\,Z + Z^3 + Z\,X\,Z + X\,Z\boxed{Y\,Z}\,Z \overset{(2)}{\downarrow =}$$

$$X\,Z + Z^2 + Y\,X\,Z + Z^3 + \boxed{Z\,X}\,Z + X\,Z^3 + X\,Z\,X\,Z \overset{(3)}{\downarrow =}$$

$$2\,X\,Z + \boxed{Y\,Z} + Z^2 + Y\,X\,Z + Z^3 + X\,Z^3 + X\,Z\,X\,Z \overset{(2)}{\downarrow =}$$

$$X + Z + 2\,X\,Z + Z^2 + Y\,X\,Z + Z^3 + X\,Z^3 + X\boxed{Z\,X}Z \overset{(3)}{=}$$

$$X + Z + 2\,X\,Z + Z^2 + X^2\,Z + \boxed{X\,Y}Z + Y\,X\,Z + Z^3 + X\,Z^3 \overset{(1)}{=}$$

$$X + Z + 2\,X\,Z + \boxed{Y\,Z} + 2\,Z^2 + X^2\,Z + Y\,X\,Z + Z^3 + X\,Z^3 \overset{(2)}{=}$$

$$2\,X + 2\,Z + 2\,X\,Z + 2\,Z^2 + X^2\,Z + Y\,X\,Z + Z^3 + X\,Z^3 \quad \square$$

$$2\,X + 2\,Z + 2\,X\,Z + 2\,Z^2 + X^2\,Z + Y\,X\,Z + Z^3 + X\,Z^3$$

The *trace* consists of a sequence of terms, each being a reduced version of the previous one. The equality used for reducing is shown above the equality sign; the initial step marked by "(…)" is just a rearrangement of the input term so that one can see how a certain rewrite rule affects it. Furthermore, we can see that the redexes are framed in each step so that we have an easy time following the whole computation. The end of the trace is marked by □ just as with proofs. The last line after the final trace term is the result of the computation; unlike all the previous text lines it is available as a regular *Mathematica* expression. This means for example that we can assign it to a variable like this:

```
result = Compute[X Y Z X,
    using → System["Test Equalities"]]
```

We compute:

$$X\,Y\,Z\,X \overset{(\ldots)}{=}$$

$$\boxed{X\,Y}Z\,X \overset{(1)}{=}$$

$$\boxed{Y\,Z}X + Z^2\,X \overset{(2)}{=}$$

$$X^2 + \boxed{Z\,X} + Z^2\,X \overset{(3)}{=}$$

$$X + Y + X^2 + Z\boxed{Z\,X} \overset{(3)}{=}$$

$$X + Y + X^2 + \boxed{Z\,X} + Z\,Y \overset{(3)}{=}$$

$$2\,X + 2\,Y + X^2 + Z\,Y \quad \square$$

$$2\,X + 2\,Y + X^2 + Z\,Y$$

Now the variable **result** contains the polynomial $2\,X + 2\,Y + X^2 + Y\,Z$, which we could use in subsequent calculations like this:

```
Compute[X (result – 2 X – 2 Y) Y,
  using → System["Test Equalities"]]
```

We compute:

$$X\,(2\,X + 2\,Y + X^2 + Z\,Y - 2\,X - 2\,Y)\,Y \overset{(...)}{=}$$

$$X^2\,\boxed{X\,Y} + X\,Z\,Y^2 \overset{(1)}{=}$$

$$X\,\boxed{X\,Y} + X^2\,Z + X\,Z\,Y^2 \overset{(1)}{=}$$

$$\boxed{X\,Y} + X\,Z + X^2\,Z + X\,Z\,Y^2 \overset{(1)}{=}$$

$$Y + Z + X\,Z + X^2\,Z + X\,Z\,Y^2 \qquad \square$$

$$Y + Z + X\,Z + X^2\,Z + X\,Z\,Y^2$$

Before passing to the more advanced features of the polynomial reductor, let us just mention some minor options that influence the formatting of the resulting polynomial and the generated trace information. We can do this systematically by looking at the following subset of **ReduceNoncommutativePolynomial** options:

```
Options[ReduceNoncommutativePolynomial]

{Indeterminates → {X, Y, Z}, Units → {}, ReductionPhases → Automatic,
  HiddenReductions → Automatic, TermOrdering → Ascending, inNotebook → Current,
  FrameRedex → True, CompactLabels → True, TraceCaption → We compute:}

?? TermOrdering

☑ Option of ReduceNoncommutativePolynomial, with default setting "Ascending", specifying an ascending
    graded term ordering. Other possible settings are "Descending" and "Lexicographic", specifying a
    descending graded and a purely lexicographic term ordering, respectively. In all cases, the ordering
    of the indeterminates is taken from the order in which they appear in the option "Indeterminetes".

?? inNotebook

☑ An option for StaticWriter that specifies, in which notebook the pretty-print output should go.

?? FrameRedex

☑ Option of ReduceNoncommutativePolynomial, with default setting True. This option determines
    whether or not the redex of polynomial forms in a trace appear with a frame around them.

?? CompactLabels

☑ Option of ReduceNoncommutativePolynomial, with default setting True. This option determines whether
    equation labels are displayed more compactly. If set to True, the environment part of the label
    is cut off, leaving only the pure equation label, which is placed above the equality symbol.
    Otherwise, the long label is placed alongside each polynomial form in the equation chain.

?? TraceCaption

☑ Option of ReduceNoncommutativePolynomial, with default setting "We compute:". This
    option specifies the string used as a caption in the trace provided for a reduction process.
```

As the online documentation listed above contains already all the essential explanations, it will be sufficient to just demonstrate these options in a few examples. First of all, let us switch off tracing and change the *term ordering* to a descending and to a lexicographic one:

```
Compute[X Y Z X,
  using → System["Test Equalities"],
  EvaluatorOptions → {TermOrdering → "Descending", inNotebook → "None"}]
```

$$X^2 + Z\,Y + 2\,X + 2\,Y$$

```
Compute[X Y Z X,
  using → System["Test Equalities"],
  EvaluatorOptions → {TermOrdering → "Lexicographic", inNotebook → "None"}]
```

$$X^2 + 2\,X + 2\,Y + Z\,Y$$

The next option mentioned above was already used in the examples: It is for controlling *where the trace goes to*. The default behavior is to produce a new notebook—just as for proofs made by the **Prove** command. By setting this option to "Current", the trace is put into the current notebook; by setting it to "None", it is simply discarded. Finally, one may set it to the name of some external file for piping the output into it. See the general *Theorema* documentation for more information.

Some people may find it too much seeing all these *frames around the redexes*. They can switch it off in the following way:

```
Compute[X Y Z X,
  using → System["Test Equalities"], EvaluatorOptions → {FrameRedex → False}]
```

We compute:

$$\mathrm{X\,Y\,Z\,X} \overset{(\ldots)}{=}$$

$$\mathrm{X\,Y\,Z\,X} \overset{(1)}{=}$$

$$\mathrm{Y\,Z\,X} + \mathrm{Z^2\,X} \overset{(2)}{=}$$

$$\mathrm{X^2} + \mathrm{Z\,X} + \mathrm{Z^2\,X} \overset{(3)}{=}$$

$$\mathrm{X} + \mathrm{Y} + \mathrm{X^2} + \mathrm{Z\,Z\,X} \overset{(3)}{=}$$

$$\mathrm{X} + \mathrm{Y} + \mathrm{X^2} + \mathrm{Z\,X} + \mathrm{Z\,Y} \overset{(3)}{=}$$

$$2\,\mathrm{X} + 2\,\mathrm{Y} + \mathrm{X^2} + \mathrm{Z\,Y} \qquad \square$$

$$2\,X + 2\,Y + X^2 + Z\,Y$$

The option about compact labels has to do with the labels set above the equality signs in the trace. The point is that although it is very convenient to see only compact labels like "(1)" there, this may lose important information sometimes. The full name of the label would be (**System** (*"Test Equalities"*): 1), which does not look very good above the equality sign. It may be necessary, however, if one has two polynomial systems, each of them containing labels with the names "(1)". Usually, it is better to avoid such duplications, but if one insists on them, one may deactivate the option for compact labels thus:

```
Compute[X Y Z X,
  using → System["Test Equalities"], EvaluatorOptions → {CompactLabels → False}]
```

We compute:

$$\mathrm{X\,Y\,Z\,X} \;\triangleright\; [\text{by } (\ldots)]$$

$$\boxed{\mathrm{X\,Y}}\,\mathrm{Z\,X} \;\triangleright\; [\text{by } (\text{Test Equalities}: 1)]$$

$\boxed{YZ}\,X + Z^2\,X \;\triangleright$ [by (Test Equalities : 2)]

$X^2 + \boxed{ZX} + Z^2\,X \;\triangleright$ [by (Test Equalities : 3)]

$X + Y + X^2 + Z\boxed{ZX} \;\triangleright$ [by (Test Equalities : 3)]

$X + Y + X^2 + \boxed{ZX} + ZY \;\triangleright$ [by (Test Equalities : 3)]

$2\,X + 2\,Y + X^2 + ZY \quad \square$

$2\,X + 2\,Y + X^2 + ZY$

Finally, let us mention the option setting the *trace caption*. This is only a minor detail whose significance will become clear in the description of the confluence tools provided below. The trace caption is simply the introductory sentence at the beginning of a trace; per default, it reads "We compute:" as one can see in all the examples above. Let us change it to something slightly more exciting, just for a whimsical test:

```
Compute[X Y Z X,
  using → System["Test Equalities"],
  EvaluatorOptions → {TraceCaption → "Hey, this is really easy for me; I just go as follows:"}]
```

Hey, this is really easy for me; I just go as follows:

$XYZX \overset{(\ldots)}{\underset{\downarrow}{=}}$

$\boxed{XY}ZX \overset{(1)}{\underset{\downarrow}{=}}$

$\boxed{YZ}\,X + Z^2\,X \overset{(2)}{\underset{\downarrow}{=}}$

$X^2 + \boxed{ZX} + Z^2\,X \overset{(3)}{\underset{\downarrow}{=}}$

$X + Y + X^2 + Z\boxed{ZX} \overset{(3)}{\underset{\downarrow}{=}}$

$X + Y + X^2 + \boxed{ZX} + ZY \overset{(3)}{\underset{\downarrow}{=}}$

$2\,X + 2\,Y + X^2 + ZY \quad \square$

$2\,X + 2\,Y + X^2 + ZY$

Let us now proceed to a much more interesting feature, concerning the usage of *parametrized indeterminates* and thereby noncommutative polynomial rings with infinitely many indeterminates. As we have seen in Chapter 1, one often needs a whole series of indeterminates like the powers $X^0, X^1, X^2, X^3, \ldots$ or the multiplication operators M_f, alternatively written as $\lceil f \rceil$, for each f in an analytic algebra. We will start with the first example, because it is easier to survey the parameter space $\mathbb{N}$. In order to avoid confusion with actual powers like X^2 occurring in the trace above, we will start with another example that uses subscripts rather than superscripts; we will later return to power notation. We will consider the noncommutative polynomial ring generated by the indeterminates $X_1, X_2, X_3, \ldots, X_\infty$ We communicate these parametrized indeterminates to the reductor in

the following way (the square is entered as ⎕sq⎕ and serves as a kind of template for the parameters to be filled in):

```
SetOptions[ReduceNoncommutativePolynomial,
  Indeterminates → {X□}];
```

Now we will set up a *simple polynomial equation* that mimics the behavior of X_n as the multiplication operator induced by $x \mapsto 10^{x/n}$. For this purpose, we must actually specify a whole series of equalities, parametrized by the subscripts ranging over 1, 2, 3, …, ∞. (Note that *Theorema* does not use explicit typing, so the domain for n and m is not stated. The implicit assumption is that it will only be used in the range intended.)

```
Formula["Subscript Equalities", any[m, n],
  X∞ = 1                  "1"
  Xm Xn = X(m+n)/(mn)     "2"]
Compute[X3 X2 X∞ X10,
  using → Formula["Subscript Equalities"],
  EvaluatorOptions → {FrameRedex → False}]
```

We compute:

$$X_3\ X_2\ X_\infty\ X_{10} \overset{(\ldots)}{=}$$

$$X_3\ X_2\ X_\infty\ X_{10} \overset{(1)}{=}$$

$$X_3\ X_2\ X_{10} \overset{(2)}{=}$$

$$X_{\frac{3+2}{3\,2}}\ X_{10} \overset{(2)}{=}$$

$$X_{\frac{\frac{3+2}{3\,2}+10}{\frac{3+2}{3\,2}\,10}} \quad \square$$

$$X_{\frac{\frac{3+2}{3\,2}+10}{\frac{3+2}{3\,2}\,10}}$$

The computation is obviously correct, but of course one would expect a bit more. It should *evaluate the subscript* $(3-12)/(3*(-12))$ to $1/4$ etc, so why is this not done? The answer is simply that it was not allowed to use any knowledge about the symbols + and − occurring in subscripts. In *Theorema*, the general assumption is that all knowledge to be exploited must be explicitly specified. In this way, we can always be sure that no assumptions are used unexpectedly so that all proofs and computations are correct with respect to the explicitly given knowledge base. In the case of the polynomial reductor, the only internal knowledge is the axioms of noncommutative polynomials (see the Appendix); all other knowledge is considered external and must hence be specified.

In *Theorema*, the usual way of doing this is via so-called *built-ins*; see the online documentation for details. In our case, the most efficient way is to rewrite the subscript equalities in terms of a new operation ∘ denoting the "harmonic sum" of two nonzero naturals:

```
Formula["Subscript Equalities", any[m, n],
  X∞ = 1          "1"
  Xm Xn = Xm∘n    "2"]
```

Then we implement the operation $\circ$ in the obvious way by using the *arithmetic operations* of *Mathematica,* and we package it into a special *Theorema* environment signified by the environment keyword **Built-in** thus:

```
HarmonicSum[m_, n_] := 1 / (1/m + 1/n)
Built-in["Arithmetic",
  ∘ → HarmonicSum]
```

Now we can compute the above polynomial in the way we expected it. Note that we have to specify built-ins via the option **built-in** rather than **using,** which we still apply to the normal knowledge base containing the formula with the subscript equalities. The difference between these two options is that the former calls *Mathematica* implementations, whereas the latter is only used in the sense of predicate logic (in the case of equalities as here, this boils down to substitution and replacement). Let us see what we get:

```
Compute[X_3 X_2 X_∞ X_10,
  using → Formula["Subscript Equalities"], built-in → Built-in["Arithmetic"]]
```

We compute:

$X_3\ X_2\ X_\infty\ X_{10} \overset{(\ldots)}{=}$

$X_3\ X_2\ \boxed{X_\infty}\ X_{10} \overset{(1)}{=}$

$\boxed{X_3\ X_2}\ X_{10} \overset{(2)}{=}$

$\boxed{X_{\frac{6}{5}}\ X_{10}} \overset{(2)}{=}$

$X_{\frac{15}{14}}$ □

$X_{\frac{15}{14}}$

This time we are satisfied with the result. So let us now look at the *case of powers*, as announced before. We have already encountered a noncommutative polynomial ring containing $X^0, X^1, X^2, X^3, \ldots$ when producing the Legendre polynomial in Computation 9 of Chapter 1. Let us study this is in a bit more detail. The first point to note is that one should not confuse the parametrized indeterminate X^2 with the product $X\,X$ of two instances of the indeterminate X. In order to communicate this difference to the reductor, we have to issue the following command:

```
UsePowers[X]
```

This tells the reductor to use the indeterminate X as a power, meaning that X^2 will not be interpreted as $X\,X$. Now we introduce the *superscripted variable* X as an indeterminate. We can compute with the polynomials of $\mathbb{C}\langle X^i \mid i \in \mathbb{N}\rangle$ as expected, but we must be very careful to understand the power notation in the appropriate way.

```
Compute[(X^2 + 1)^2,
  EvaluatorOptions → {Indeterminates → {X^□}, inNotebook → "None"}]
```

$1 + 2\,X^2 + X^{2\,2}$

Here the monomial X^{2^2} is a shortcut for $X^2\,X^2$. *Internally*, this is clearly distinguished as one can see using the in the *Mathematica* **InputForm**:

```
InputForm[%]

TM Plus[1, TM Times[2, TM PseudoPower[X, 2]],
 TM Power[TM PseudoPower[X, 2], 2]]
```

This means that the given polynomial is a sum (TM Plus) having the summands 1 and TM Times[2, TM PseudoPoser[X, 2]] and TM Power[TM PseudoPower[X, 2], 2]. The former represents a product (TM Times) of 2 and TM PseudoPower[X, 2], whereas the latter is the square (TM Power[…, 2]) of TM PseudoPoser[X, 2]. So we can see that the indeterminate X^2 has the internal representation TM PseudoPower[X, 2]. We have called these expressions *pseudo powers*, because it looks like a power but it is not.

Note that the internal name of all symbols within *Theorema* formulae have a leading TM in order to keep them out of the *Mathematica namespace*. For example, the symbol + is TM Plus in a *Theorema* formula but Plus otherwise. We can see this in the following example, where we compare the first formula of the test equalities considered above with the analogous *Mathematica* expression:

```
System["Test Equalities"][[4, 1, 2]] // InputForm

TM Equal[TM Times[X, Y], TM Plus[Y, Z]]

FullForm[X Y == Y Z]

Equal[Times[X, Y], Times[Y, Z]]
```

For making the difference even clearer let us do the *same computation as before* in the polynomial ring $\mathbb{C}\langle X\rangle \cong \mathbb{C}[X]$. We undo the declaration for using power notation, and then we start the computation with the corresponding setting for the indeterminates.

```
DoNotUsePowers[X]

Compute[(X^2 + 1)^2,
  EvaluatorOptions → {Indeterminates → {X}, inNotebook → "None"}]

1 + 2 X^2 + X^4
InputForm[%]

TM Plus[1, TM Times[2, TM Power[X, 2]], TM Power[X, 4]]
```

Now the result is different, because X^{2^2} was understood as TM Power[TM Power[X, 2], 2], which gives of course TM Power[X, 4], where TM Power is the *real power* (rather than the pseudo power) used for abbreviating iterated multiplication.

As a next step, we want to *mimic the algebraic laws* used above for reducing $(X^2+1)^2$ to $1+2\,X^2+X^4$ so that we get an analogous result in the polynomial ring $\mathbb{C}\langle X^n \mid n \in \mathbb{N}\rangle$. But before doing so, let us ask ourselves why we would take these troubles. Why not simply use the real powers as above? The answer is that the commutation between multiples of X and the differentiation indeterminate D is not be very efficient in this setup. Of course, one *can* do it using the following simplistic product rule:

```
Formula["Simplistic Product Rule",
  D X = X D + 1   "DX" ]

Compute[D X^6,
  using → Formula["Simplistic Product Rule"],
  EvaluatorOptions → {Indeterminates → {D, X}}]
```

We compute:

$$D\,X^6 \overset{(\ldots)}{=}$$

$$\boxed{D\,X}\,X^5 \overset{(DX)}{=}$$

$$X^5 + X\,\boxed{D\,X}\,X^4 \overset{(DX)}{=}$$

$$2\,X^5 + X^2\,\boxed{D\,X}\,X^3 \overset{(DX)}{=}$$

$$3\,X^5 + X^3\,\boxed{D\,X}\,X^2 \overset{(DX)}{=}$$

$$4\,X^5 + X^4\,\boxed{D\,X}\,X \overset{(DX)}{=}$$

$$5\,X^5 + X^5\,\boxed{D\,X} \overset{(DX)}{=}$$

$$6\,X^5 + X^6\,D \qquad \square$$

$$6\,X^5 + X^6\,D$$

Obviously this is not a very useful way of handling the product rule. So having the atomic indeterminate X makes multiplication trivial, but only at the cost of making differentiation complicated. Therefore let us move to the ring $\mathbb{C}\langle X^n \mid n \in \mathbb{N}\rangle$ again, and let us fix the $X^{\square}$ *family of indeterminates* together with the differentiation indeterminate D.

```
UsePowers[X]
SetOptions[ReduceNoncommutativePolynomial,
    Indeterminates → {D, X^□}];
Formula["Product Rule", any[n],
  D X^n = X^n D + n X^(n⊖1)   "DX" ]
```

As explained before, we must explicitly state all external knowledge. That is why we have used the symbol $\ominus$ for denoting subtraction on the natural numbers. We must provide its implementation, and the most natural choice is of course the *Mathematica* command Minus. Having done so, the computation from above does indeed become a trivial *one-step shot* as expected:

```
Built-in["Arithmetic",
  ⊖ → Minus]
SetOptions[ReduceNoncommutativePolynomial,
    inNotebook → "None"];
Compute[D X^6,
  using → Formula["Product Rule"], built-in → Built-in["Arithmetic"]]
```

$$6\,X^5 + X^6\,D$$

But now *multiplication* is of course not possible any more:

```
Compute[X^2 X^3,
  using → Formula["Product Rule"], built-in → Built-in["Arithmetic"],
  EvaluatorOptions → {inNotebook → "None"}]
```

$$X^2\,X^3$$

Unless we provide the corresponding laws:

```
Formula["Power Law", any[m, n],
  X^m X^n = X^(m⊕n)   "XX" ]
```

Here we have used the symbol $\oplus$ for denoting addition on the natural numbers. We extend the implementation of arithmetic in the obvious way, and the computation from above becomes indeed feasible.

```
Built-in["Arithmetic",
  ⊕ → Plus
  ⊖ → Minus]
SetOptions[Compute,
  using → {Formula["Power Law"], Formula["Product Rule"]}, built-in → Built-in["Arithmetic"]];
Compute[X^2 X^3]
```

X^5

We have *forgotten one more equality* needed in this context: What happens when $D\,X$ is reduced? The product rule will produce a "power" X^0, which cannot be recognized as being the same as 1 if it stands by itself:

```
Compute[D X]
```

$X^0 + X\,D$

We could fix this problem by adding the equality $X^0 = 1$. But there is a shortcut to this: Since equalities saying that some indeterminate is equal to 1 for a certain choice of the parameter, there is a special option telling the reductor that such an indeterminate acts as a *unit*. We will add this option globally, since we will need it again afterwards:

```
?? Units
```

Option of ReduceNoncommutativePolynomial, with default setting {}. The option value is interpreted as a list of polynomial forms to be regarded as equal to 1. For example, setting Units→{X^0} makes X^0 reduce to 1 anywhere during the rewriting process. In effect, this options amounts to adding the corresponding equation X^0=1 to the ubiquitous hidden knowledge base.

☑

```
SetOptions[ReduceNoncommutativePolynomial,
   Units → {X^0}];
Compute[D X]
```

$1 + X\,D$

The option **Units** takes a list of *specialized indeterminates* that are to be considered as units. The corresponding reductions like $X^0 \to 1$ are not traced, however, just like hidden reduction rules (see the explanation below).

We have not yet considered an example where *both formulae*—the product rule and the power law—are needed. But there are of course plenty of such examples, and the Legendre operator of Computation 9 of Chapter 1 are a case in point. Let us here look at a slightly more representative example, which necessitates several XX steps between the DX steps. For this purpose, we reactivate the tracing option globally, and we set two other options locally which we will explain immediately.

```
SetOptions[ReduceNoncommutativePolynomial,
   inNotebook → "Current"];
Compute[D X^2 D^2 X^3,
  EvaluatorOptions → {ReductionPhases → {}, HiddenReductions → {}}]
```

We compute:

$$D\,X^2\,D^2\,X^3 \overset{(\ldots)}{=}$$

$$\boxed{D\,X^2}\,D^2\,X^3 \overset{(DX)}{=}$$

$$2\,X\,D\,\boxed{D\,X^3} + X^2\,D^3\,X^3 \overset{(DX)}{=}$$

$$6\,X\,\boxed{D\,X^2} + 2\,X\,D\,X^3\,D + X^2\,D^3\,X^3 \overset{(DX)}{=}$$

$$12\,\boxed{X^2} + 6\,X\,X^2\,D + 2\,X\,D\,X^3\,D + X^2\,D^3\,X^3 \overset{(XX)}{=}$$

$$12\,X^2 + 6\,\boxed{X\,X^2}\,D + 2\,X\,D\,X^3\,D + X^2\,D^3\,X^3 \overset{(XX)}{=}$$

$$12\,X^2 + 6\,X^3\,D + 2\,X\,\boxed{D\,X^3}\,D + X^2\,D^3\,X^3 \overset{(DX)}{=}$$

$$12\,X^2 + 6\,X^3\,D + 6\,\boxed{X\,X^2}\,D + 2\,X\,X^3\,D^2 + X^2\,D^3\,X^3 \overset{(XX)}{=}$$

$$12\,X^2 + 12\,X^3\,D + 2\,\boxed{X\,X^3}\,D^2 + X^2\,D^3\,X^3 \overset{(XX)}{=}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + X^2\,D^2\,\boxed{D\,X^3} \overset{(DX)}{=}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + 3\,X^2\,D\,\boxed{D\,X^2} + X^2\,D^2\,X^3\,D \overset{(DX)}{=}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + 6\,X^2\,\boxed{D\,X} + 3\,X^2\,D\,X^2\,D + X^2\,D^2\,X^3\,D \overset{(DX)}{=}$$

$$12\,X^2 + 12\,X^3\,D + 6\,\boxed{X^2\,X^0} + 2\,X^4\,D^2 + 6\,X^2\,X\,D + 3\,X^2\,D\,X^2\,D + X^2\,D^2\,X^3\,D \overset{(XX)}{=}$$

$$18\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + 6\,\boxed{X^2\,X}\,D + 3\,X^2\,D\,X^2\,D + X^2\,D^2\,X^3\,D \overset{(XX)}{=}$$

$$18\,X^2 + 18\,X^3\,D + 2\,X^4\,D^2 + 3\,X^2\,\boxed{D\,X^2}\,D + X^2\,D^2\,X^3\,D \overset{(DX)}{=}$$

$$18\,X^2 + 18\,X^3\,D + 2\,X^4\,D^2 + 6\,\boxed{X^2\,X}\,D + 3\,X^{2\,2}\,D^2 + X^2\,D^2\,X^3\,D \overset{(XX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 2\,X^4\,D^2 + 3\,\boxed{X^{2\,2}}\,D^2 + X^2\,D^2\,X^3\,D \overset{(XX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 5\,X^4\,D^2 + X^2\,D\,\boxed{D\,X^3}\,D \overset{(DX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 5\,X^4\,D^2 + 3\,X^2\,\boxed{D\,X^2}\,D + X^2\,D\,X^3\,D^2 \overset{(DX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 5\,X^4\,D^2 + 6\,\boxed{X^2\,X}\,D + 3\,X^{2^2}\,D^2 + X^2\,D\,X^3\,D^2 \overset{(XX)}{\downarrow =}$$

$$18\,X^2 + 30\,X^3\,D + 5\,X^4\,D^2 + 3\,\boxed{X^{2^2}}\,D^2 + X^2\,D\,X^3\,D^2 \overset{(XX)}{\downarrow =}$$

$$18\,X^2 + 30\,X^3\,D + 8\,X^4\,D^2 + X^2\,\boxed{D\,X^3}\,D^2 \overset{(DX)}{\downarrow =}$$

$$18\,X^2 + 30\,X^3\,D + 8\,X^4\,D^2 + 3\,\boxed{X^{2^2}}\,D^2 + X^2\,X^3\,D^3 \overset{(XX)}{\downarrow =}$$

$$18\,X^2 + 30\,X^3\,D + 11\,X^4\,D^2 + \boxed{X^2\,X^3}\,D^3 \overset{(XX)}{\downarrow =}$$

$$18\,X^2 + 30\,X^3\,D + 11\,X^4\,D^2 + X^5\,D^3 \quad \square$$

$$18\,X^2 + 30\,X^3\,D + 11\,X^4\,D^2 + X^5\,D^3$$

Looking at the trace above, we have the feeling that the "boring" XX steps are somehow distracting from the "actually important" DX steps; and this becomes even more prominent when considering the full set of Green's identities explicated repeatedly in Chapter 1. Somehow one considers the steps for contracting $X^m\,X^n$ to $X^{m\oplus n}$ as a low-level operation that is not in focus here; in fact, we introduced it only for technical reasons as explained above. That is why the evaluator provides an option for hiding such uninteresting steps, which are called *hidden reductions*. In the **Compute** call above, we have explicitly asked for having no hidden reductions by setting the option **HiddenReductions** to the value {}. Let us now set it to the reductions specified in the formula with the label *"Product Law"*. Naturally, this is done by listing just the string containing the label in the list that was formerly empty (we assume that the string is not reused across different environment labels in one and the same call as this would be really bad style):

```
Compute[D X^2 D^2 X^3,
  EvaluatorOptions → {ReductionPhases → {}, HiddenReductions → {"Power Law"}}]
```

We compute:

$$D\,X^2\,D^2\,X^3 \overset{(...)}{\downarrow =}$$

$$\boxed{D\,X^2}\,D^2\,X^3 \overset{(DX)}{\downarrow =}$$

$$2\,X\,D\,\boxed{D\,X^3} + X^2\,D^3\,X^3 \overset{(DX)}{\downarrow =}$$

$$6\,X\,\boxed{D\,X^2} + 2\,X\,D\,X^3\,D + X^2\,D^3\,X^3 \overset{(DX)}{\downarrow =}$$

$$12\,X^2 + 6\,X^3\,D + 2\,X\,\boxed{D\,X^3}\,D + X^2\,D^3\,X^3 \overset{(DX)}{\downarrow =}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + X^2\,D^2\,\boxed{D\,X^3} \overset{(DX)}{\downarrow =}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + 3\,X^2\,D\,\boxed{D\,X^2} + X^2\,D^2\,X^3\,D \overset{(DX)}{\downarrow =}$$

$$12\,X^2 + 12\,X^3\,D + 2\,X^4\,D^2 + 6\,X^2\,\boxed{D\,X} + 3\,X^2\,D\,X^2\,D + X^2\,D^2\,X^3\,D \overset{(DX)}{=}$$

$$18\,X^2 + 18\,X^3\,D + 2\,X^4\,D^2 + 3\,X^2\,\boxed{D\,X^2}\,D + X^2\,D^2\,X^3\,D \overset{(DX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 5\,X^4\,D^2 + X^2\,D\,\boxed{D\,X^3}\,D \overset{(DX)}{=}$$

$$18\,X^2 + 24\,X^3\,D + 5\,X^4\,D^2 + 3\,X^2\,\boxed{D\,X^2}\,D + X^2\,D\,X^3\,D^2 \overset{(DX)}{=}$$

$$18\,X^2 + 30\,X^3\,D + 8\,X^4\,D^2 + X^2\,\boxed{D\,X^3}\,D^2 \overset{(DX)}{=}$$

$$18\,X^2 + 30\,X^3\,D + 11\,X^4\,D^2 + X^5\,D^3 \qquad \square$$

$$18\,X^2 + 30\,X^3\,D + 11\,X^4\,D^2 + X^5\,D^3$$

Now this is much more compact to read; the trivial "algebraic" steps are suppressed. Let us now add *two more indeterminates* L, R denoting left and right boundary values on the interval $[0, 1]$.

```
SetOptions[ReduceNoncommutativePolynomial,
    Indeterminates → {D, X^□, L, R}];
```

A typical collection of *identities for dealing with these boundary operators* is as following. The differentiation of any constant gives zero, hence $D\,L$ and $D\,R$ does as well; we add these identities to the product rule, because they are all instances of the more general law of differentiating "across a function" (which is either a simple power as in X^n or a constant function as in L and R). Furthermore, the left and right boundary value of $x \mapsto x^n$ is 0 and 1, respectively, hence we have $L\,X^n = 0$, $R\,X^n = X^n$; we will package this in a system by itself.

```
System["Product Laws", any[n],
  D X^n = X^n D + n X^(n⊖1)   "DX"
  D L = 0                     "DL" ]
  D R = 0                     "DR"
System["Boundary Laws", any[n],
  L X^n = 0   "LX"
  R X^n = R   "RX"
  L L = L     "LL"
  L R = R     "LR" ]
  R L = L     "RL"
  R R = R     "RR"
```

Now we could of course throw all of these identities in one big pot, and for those few identities considered here it would actually not do much harm. But for enhancing clarity as well as efficiency, it is more advisable—especially for bigger systems like the Green's identities—to collect the identities into blocks that are to be used in succeeding *reduction phases*. We have explained this in some detail after Convention 13 in Chapter 1. Even in the toy example considered here, it is still reasonable to apply the product laws in a first phase (with the "operational goal" of pushing the differential operators to the very right) and the boundary laws in a second phase (with the "operational goal" of pushing the boundary operators as far right as possible); the "algebraic" reductions due to the power law should of course be applied throughout. We call equalities belong-

ing to a certain phase (like the product and boundary laws in our example) "phase equalities" and those to be applied throughout "ubiquitous equalities" (like the power law in our example). In the **Compute** call, this is handled via the option **ReductionPhases**, which takes the list of all the labels belonging to phase equalities. In our example, we would thus say (making the ubiquitous reductions from the power law hidden as before):

```
SetOptions[Compute,
  using → {System["Product Laws"], System["Boundary Laws"], Formula["Power Law"]}];
SetOptions[ReduceNoncommutativePolynomial,
  HiddenReductions → {"Power Law"}, ReductionPhases → {"Product Laws", "Boundary Laws"}];
```

Note that the labels in the list after **ReductionPhases** are processed in the order they are given, meaning that *"Product Laws"* makes up the first phase and *"Boundary Laws"* the second. Of course, some labels might also repeat, if a certain phase should be entered more than once. Incidentally, we could have left out the specification of the hidden reductions above, because the default setting is to hide exactly the ubiquitous reductions as this is the usual situation. This is also explained in the official online documentation:

```
?? HiddenReductions
```

Option of ReduceNoncommutativePolynomial, with default setting Automatic. The
☑ option value is interpreted as a list of labels referring to those environments that should be used without tracing. The automatic setting means that exactly the ubiquitous reductions are hidden – those polynomial rules that are not specified as reduction phases.

```
?? ReductionPhases
```

Option of ReduceNoncommutativePolynomial, with default setting Automatic. The option value is
☑ interpreted as a list of labels referring to those environments that should be used as subsequent reduction phases in the order specified by this option. The automatic setting means that all the environments are taken as reduction phases in the order in which they are specified.

Before concluding the description of the reductor and its options, let us do a *small computation* in the setup introduced above.

```
Compute[R X^2 (D X^3 R X^2 - R (X^2 + 1)) + L D X]
```

We compute:

$$R\,X^2\,(D\,X^3\,R\,X^2 - R\,(X^2+1)) + L\,D\,X \overset{(\ldots)}{\downarrow=}$$

$$L\,\boxed{D\,X} - R\,X^2\,R - R\,X^2\,R\,X^2 + R\,X^2\,D\,X^3\,R\,X^2 \overset{(DX)}{\downarrow=}$$

$$L + L\,X\,D - R\,X^2\,R - R\,X^2\,R\,X^2 + R\,X^2\,\boxed{D\,X^3}\,R\,X^2 \overset{(DX)}{\downarrow=}$$

$$L + L\,X\,D - R\,X^2\,R - R\,X^2\,R\,X^2 + 3\,R\,X^4\,R\,X^2 + R\,X^5\,\boxed{D\,R}\,X^2 \overset{(DR)}{\downarrow=}$$

$$L + \boxed{L\,X}\,D - R\,X^2\,R - R\,X^2\,R\,X^2 + 3\,R\,X^4\,R\,X^2 \overset{(LX)}{\downarrow=}$$

$$L - \boxed{R\,X^2}\,R - R\,X^2\,R\,X^2 + 3\,R\,X^4\,R\,X^2 \overset{(RX)}{\downarrow=}$$

$$L - R^2 - \boxed{R\,X^2}\,R\,X^2 + 3\,R\,X^4\,R\,X^2 \overset{(RX)}{\downarrow=}$$

$$L - R^2 - R\boxed{R\,X^2} + 3\,R\,X^4\,R\,X^2 \overset{(RX)}{\downarrow} =$$

$$L - 2\,R^2 + 3\boxed{R\,X^4}R\,X^2 \overset{(RX)}{\downarrow} =$$

$$L - 2\,R^2 + 3\,R\boxed{R\,X^2} \overset{(RX)}{\downarrow} =$$

$$L + \boxed{R^2} \overset{(RR)}{\downarrow} =$$

$$L + R \qquad \square$$

$$L + R$$

This finishes the presentation of the evaluator itself. As mentioned above, there is also a small add-on toolbox, which we have called the *confluence tools* (comprising the function described here and that described in Section 5), because they can help proving the confluence of some reduction systems; and this is of course what we did in the proof of Theorem 28 in Chapter 1. Note that we do not regard these tools as a ready-to-go package like the reductor or the Green's evaluator—it was only designed for supporting this proof. Hence its usage is somewhat more technical and demands some knowledge of *Mathematica* and *Theorema* programming. The way the function described here was actually used is only through a special interface function named **ProveConfluence** within the Green's evaluator, which will be described in Section 5. The idea of the **ProveConfluence** function is to adapt the generic computation of S-polynomials to the special situation of the Green's identities.

The main function to be considered here is called **SPolynomials**, since it determines all the *rule overlaps* and computes their S-polynomials. The interface to this function is very similar to that of the basic evaluator **ReduceNoncommutativePolynomial** itself; in fact, it follows the standard of all *Theorema* provers and evaluators: It accepts a visible, a hidden, and a built-in database as well as some options that are identical with those of **ReduceNoncommutativePolynomial**.

Note that for provers and evaluators, the user normally does not see their own *interface*, because they are only called indirectly through the **Prove** and **Compute** functions. Such a generic procedure would not be appropriate for determining the S-polynomials, though, and that is why for the moment we prefer to address the function **SPolynomials** by its own interface or rather by the function **ProveConfluence** described in Section 5. In a later version of *Theorema* we might have a clear standard for such "extra calculations", much in the sense of integrating proving and computing as explained in the Section 1.

But for getting some basic understanding about its functionality, let us prove confluence for the toy example above using only the function **SPolynomials** directly. The most important point is to generate the *knowledge base* to be used, in our case containing the power, product and boundary laws. The most convenient way to do so is via the global *Theorema* knowledge base. First we clear the knowledge base (it should actually be empty, but we want to be sure) by the command **Use**, then we add the three environments needed using the command **UseAlso**. The result will be available via the internal *Theorema* slot **$TmaUserKB[•kb]**, so we save this value to an auxiliary variable **kb**, and then we clear up the global knowledge base again:

```
Use[]
UseAlso[Formula["Power Law"]]
UseAlso[System["Product Laws"]]
UseAlso[System["Boundary Laws"]]
```

```
kb = Theorema`Language`Semantics`UserLanguage`Private`$TmaUserKB[•kb];
Use[]
```

Now we can use the function **SPolynomials** using the auxiliary variable **kb** for the *visible knowledge* base in the first argument. We do *no hidden knowledge*, so we use the generic *Theorema* template **EmptyEnvironment** for providing an empty environment; and the list of *built-ins is empty* as well (note that, unlike the proper knowledge bases like those before, the built-ins are always stored as lists in *Theorema*). Then we feed in some options, analogous to what we would use for calling **Compute** for the reduction system under investigation.

```
SPolynomials[kb, EmptyEnvironment[•kb, "None"], {},
  Indeterminates → {D, X□, L, R}, Units → {X^0}, ReductionPhases → {}, HiddenReductions → {}]
```

$$\{\{\text{LL, LX, } L\,X^{n}\},\ \{\text{LL, LL, } 0\},\ \{\text{LL, LR, } 0\},\ \{\text{LR, RX, } -L\,R + R\,X^{n}\},\ \{\text{LR, RL, } -L^2 + R\,L\},\ \{\text{LR, RR, } -L\,R + R^2\},$$
$$\{\text{RL, LX, } L\,X^{n}\},\ \{\text{RL, LL, } L^2 - R\,L\},\ \{\text{RL, LR, } L\,R - R^2\},\ \{\text{RR, RX, } R\,X^{n} - R^2\},\ \{\text{RR, RL, } 0\},\ \{\text{RR, RR, } 0\},$$
$$\{\text{DL, LX, } 0\},\ \{\text{DL, LL, } -D\,L\},\ \{\text{DL, LR, } -D\,R\},\ \{\text{DR, RX, } -D\,R\},\ \{\text{DR, RL, } -D\,L\},\ \{\text{DR, RR, } -D\,R\}\}$$

```
spolys = Select[%[[All, 3]], # =!= 0 &]
```

$$\{L\,X^{n},\ -L\,R + R\,X^{n},\ -L^2 + R\,L,\ -L\,R + R^2,\ L\,X^{n},$$
$$L^2 - R\,L,\ L\,R - R^2,\ R\,X^{n} - R^2,\ -D\,L,\ -D\,R,\ -D\,R,\ -D\,L,\ -D\,R\}$$

The result returned by **SPolynomials** is a list containing one triple for each overlap: The first two elements of each triple signify the *rules involved* in the overlap, whereas the third is the *S-polynomial* produced from them. For example, the first triple in the list tells us that the rules LL and LX overlapped (namely on the monomial $L\,L\,X^n$), and they yield the S-polynomial $L\,X^n$ (because reducing $L\,L\,X^n$ by the rules LL and LX yields $L\,X^n$ and 0, respectively, with the difference being $L\,X^n$).

Now some of these S-polynomials came out to zero immediately, whereas the others are easily seen to reduce to zero. For doing this automatically, we produce the list of all the S-polynomials from the list of triples by **%[[All, 3]]**, and then we select the sublist of nonzero entries from the resulting list by saying **Select[…, # =!= 0 &]**. This *list of nonzero S-polynomials* is stored in an auxiliary variable **spolys**.

Finally, we reduce all of these nonzero S-polynomials by mapping the **Compute** function over them.

```
SetOptions[ReduceNoncommutativePolynomial,
    inNotebook → "None"];
Compute /@ spolys
```

$$\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$$

All of them came out to zero, as we expected. The function **ProveConfluence** to be described in Section 5 can be thought of as a kind of *elaborate interface* automating the hand-crafted procedure carried out above. Moreover, it provides some support for using axioms about the parameter domain—a crucial feature for the successful application of the confluence tools to the proof of Theorem 28 in Chapter 1.

2.4 The Matrix Evaluator

The matrix evaluator **EvaluateMatrices** provides the following (mostly partial) operations on the graded matrix ring $\bigcup_{m,n=1}^{\infty} R^{m\times n}$ over some base ring R containing $\mathbb{C}$:

- *Constructor*: If mn complex numbers $c_{11}, \ldots, c_{1n}, \ldots, c_{n1}, \ldots, c_{nn}$ are given, the $m \times n$ matrix

$$\begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \ldots & c_{nn} \end{pmatrix}$$

  may be formed in the usual *Mathematica* manner of grids, using CTRL-, for creating columns and CTRL-RET for creating rows.

- *Selector*: If M is an $m \times n$ matrix and $1 < i < m$, the selector M_i returns the i-th row of M, being itself a $1 \times n$ matrix (hence a row matrix).

- *Quantifier*: If $\mathcal{T}$ is some term typically containing a free occurrence of n, we can create a $(B - A + 1) \times 1$ (hence a column vector) with entries $\mathcal{T}_{n \leftarrow A}$, $\mathcal{T}_{n \leftarrow A+1}$, …, $\mathcal{T}_{n \leftarrow B-1}$, $\mathcal{T}_{n \leftarrow B}$ by $\langle \mathcal{T} \mid n = A, \ldots, B \rangle$. Note the similarity to the set quantifier $\{\mathcal{T} \mid n = A, \ldots, B\}$, which would collect all the entries in an unordered and repetition-free way.

- *Dimension*: If M is an $n \times n$ matrix, dim[M] is n. (If M is not a square matrix, this function should not be used; it would return the number of columns of M.)

- *Addition*: If M_1 and M_2 are both $m \times n$ matrices, their sum may simply be formed as $M_1 + M_2$. Note that $M_1 - M_2$ is of course $M_1 + (-1) M_2$, where the premultiplier -1 is to be understood as a scalation (see below).

- *Scalation*: If M is an $m \times n$ matrix with entries $m_{11}, \ldots, m_{1n}, \ldots, m_{n1}, \ldots, m_{nn}$ and c is a complex number, the matrix M may be scaled by c to a matrix having entries $cm_{11}, \ldots, cm_{1n}, \ldots, cm_{n1}, \ldots, cm_{nn}$. This matrix may be denoted by either cm or by mc.

- *Multiplication*: If M_1 is an $m \times n$ and M_2 an $n \times k$ matrix, their product may simply be specified as $M_1 M_2$. Note that the "multiplication symbol" (which can be either the explicit operator symbol $*$ or simply juxtaposition) is overloaded—it denotes ordinary multiplication if both operands are scalars, scalation if one of them is a scalar and the other one a matrix, matrix multiplication if they are both matrices.

- *Inversion*: If M is a regular $n \times n$, its inverse is denoted by M^{-1}. Note again the overloading involved in this notation; for normal complex numbers, the ordinary reciprocal is of course used instead.

Let us now do some examples. First, let us *construct* a generic 2×2 matrix and analyze its internal form:

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```
InputForm[M]
```

```
™Matrix[™Tuple[™Tuple[a, b], ™Tuple[c, d]]]
```

Obviously matrices are regarded as *tuples containing tuples of the same length*, but they are again packed into a container named **™Matrix** in order to ensure that the type information is not

lost.

Now let us select the *first row* and its *second row*.

```
SetOptions[Compute, by → EvaluateMatrices];
Compute[M1]
```

$$(a \quad b)$$

```
Compute[M2]
```

$$(c \quad d)$$

Let us now construct a column vector by the *quantifier*:

```
Compute[⟨S^n | n = 2, …, 4⟩]
```

$$\begin{pmatrix} S^2 \\ S^3 \\ S^4 \end{pmatrix}$$

We convince ourselves that M is indeed a 2×2 matrix, i.e. it has *dimension* 2:

```
Compute[dim[M]]
```

$$2$$

Let us now *add* M to itself, *triple* it, *multiply* it by itself, and finally *invert* is:

```
Compute[M + M]
```

$$\begin{pmatrix} a+a & b+b \\ c+c & d+d \end{pmatrix}$$

```
Compute[3 M]
```

$$\begin{pmatrix} 3\,a & 3\,b \\ 3\,c & 3\,d \end{pmatrix}$$

```
Compute[M M]
```

$$\begin{pmatrix} a\,a+b\,c & a\,b+b\,d \\ c\,a+d\,c & c\,b+d\,d \end{pmatrix}$$

```
Compute[M^-1]
```

$$\begin{pmatrix} d\,(-1\,b\,c+a\,d)^{-1} & -1\,b\,(-1\,b\,c+a\,d)^{-1} \\ -1\,c\,(-1\,b\,c+a\,d)^{-1} & a\,(-1\,b\,c+a\,d)^{-1} \end{pmatrix}$$

As we can see from the examples above, *no special knowledge* is assumed about the base ring R. In particular, we do not presuppose that R is commutative. This is important for using the matrix evaluator for working with operator matrices as applied in the Green's evaluator; see Section 5.

The overloading mechanism described above is trivial as long as one deals with *concrete matrices*. This was the case in all the examples above. Consider the following typical situation:

```
Compute[[(r  s)  + (1  2)]]
        [(t  u)    (3  4)]
```

$$\begin{pmatrix} r+1 & s+2 \\ t+3 & u+4 \end{pmatrix}$$

Here it is clear that the symbol + denotes matrix addition, because both the left and the right operands are readily recognized as matrices: The constructor used for building them necessarily

returns a matrix. But now let us think about the slightly *more complicated situation* of Input 43. We had the following formulae there:

Definition["*Wronski Operator*", any[n],
$\hat{D}_n = \langle D^i \mid i = 0, \ldots, n-1 \rangle$
]

Formula["*Nullspace Projector*", any[$\hat{w}$, $\hat{l}$, $\hat{r}$],
$\mathrm{Proj}_{\hat{w}}[\hat{l}, \hat{r}] = \lceil \hat{w}_1 \rceil \left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1} \left(L\,\hat{l}\,\hat{D}_n + R\,\hat{r}\,\hat{D}_n\right) \Big| [n = \dim[\hat{w}]]$
]

How can we know that the symbol + in the subterm $\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}$ is supposed to denote matrix addition, the next operation matrix inversion, etc? The answer is that we have introduced the following *typing convention*: All variables carrying a hat are considered as matrices (including row and column vectors), all other variables as scalars. Using this scheme, it is fairly straightforward to set up a corresponding type inference, which goes roughly as follows. Since $\hat{w}$ is a matrix, so are $\hat{w}^{\leftarrow}$ and $\hat{w}^{\rightarrow}$ as well as the products $\hat{l}\,\hat{w}^{\leftarrow}$ and $\hat{r}\,\hat{w}^{\rightarrow}$; hence $\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}$ is their matrix sum, being itself a matrix again. Therefore $\left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1}$ will be understood as matrix inversion, etc.

Finally, let us remark that the base ring R is identified with $R^{1\times 1}$. This saves us from the trouble of accessing the entries of such 1×1 matrices via subscripting. For example, multiplying a 1×2 with a 2×1 matrix will give a scalar:

```
a = ( r  s );
InputForm[a]
```
```
TMMatrix[TMTuple[TMTuple[r, s]]]
```
```
b = ( u
      v );
InputForm[b]
```
```
TMMatrix[TMTuple[TMTuple[u], TMTuple[v]]]
```
```
Compute[a b]
```
```
r u + s v
```
```
InputForm[%]
```
```
TMPlus[TMTimes[r, u], TMTimes[s, v]]
```

We can see that the last result is not packed into **TMMatrix** anymore, meaning that it is a scalar. The effect of unpacking is seen even more clearly by using the internal forms:

```
Compute[TMMatrix[TMTuple[TMTuple[u], TMTuple[v]]]]
```
```
( u
  v )
```
```
InputForm[%]
```
```
TMMatrix[TMTuple[TMTuple[u], TMTuple[v]]]
```
```
Compute[TMMatrix[TMTuple[TMTuple[u]]]]
```
```
u
```
```
InputForm[%]
```
```
u
```

The identification of 1×1 matrices and scalars is important for the computation of the nullspace projector effected by the formula given above: It will come out as a scalar in the end for the following reason. Starting from left, $\lceil \hat{w}_1 \rceil$ is an $1 \times n$ matrix (namely the first row of the Wronskian matrix, i.e. the fundamental system written as a row vector). Then it is multiplied by the $n \times n$ matrix $\left(\hat{l}\,\hat{w}^{\leftarrow} + \hat{r}\,\hat{w}^{\rightarrow}\right)^{-1}$, thus still being of format $1 \times n$. Finally, $L\,\hat{l}\,\hat{D}_n + R\,\hat{r}\,\hat{D}_n$ has format $n \times 1$, since $\hat{D}_n$ does (the Wronski operator yields a column vector because it is defined in terms of the quantifier construct explained above) and both $\hat{l}$ and $\hat{r}$ are $n \times n$, whereas both L and R are scalars. Hence we have a product of a $1 \times n$ and an $n \times 1$ matrix, giving a 1×1 matrix that is identified with its scalar entry.

2.5 The Green's Evaluator

The *Green's evaluator* is the interface used for solving BVPs by the method explicated in Chapter 1. It is called **GreenEvaluator** and can be used in a **Compute** call just as any other evaluator. It can do various tasks according to the term transmitted:

- *Computing Nullspace Projectors*: For obtaining the nullspace projector associated with the boundary matrices $\hat{l}$ and $\hat{r}$ and the Wronskian matrix $\hat{w}$, one must compute $\text{Proj}_{\hat{w}}\left[\hat{l}, \hat{r}\right]$ with a knowledge base containing the appropriate definitions for $\hat{l}$, $\hat{r}$ and $\hat{w}$. The computation is carried out by unfolding the formulae in Input 43. See the examples given there.

- *Right Inversion*: For obtaining the right inverse of a linear constant-coefficient operator T, one must compute $T^{\blacklozenge}$ with a knowledge base containing some definition of T. The computation is carried out by unfolding the definition of Input 41. See the examples given there.

- *Green's Reduction*: The kernel of the Green's evaluator is of course the interface to the reductor, specialized to the noncommutative polynomial ring $\mathcal{A}n$ and the Green's identities of Input 14. For carrying out a reduction in this sense, one can simply feed in the polynomial to be reduced.

- *Miscellaneous Built-ins*: Some of the formulae referred to above need external operations that are delegated to *Mathematica*. The formula for the right inverse uses three functions named poly, deg, rad for computing the characteristic polynomial of a differential operator, the degree of a polynomial and the roots of a polynomial, respectively. In the formula for the Green's operator, the functions wron, left and right are used for computing the Wronskian matrix of a differential operator and the left/right boundary matrix of a system of boundary operators, respectively. All these external functions are implemented and made available in the Green's evaluator.

- *Basis Expansion*: As explained in Chapter 1 after Definition 12, analytic polynomials have to be subjected to a process of basis expansion after reduction, otherwise they will in general not remain analytic polynomials. The Green's evaluator takes care of this issue automatically.

- *Action Operators*: The analytic polynomials are built on an analytic algebra that forms the parameter domain of the multiplication operators. The operations available on this domain—differential, integral, cointegral, left and right boundary action—are fully supported by the Green's evaluator.

- *Computing Green's Operators*: This is of course the heart piece, which puts together all the components listed above. For obtaining the Green's operator induced by the differential operator T and the boundary operators $B_1, \ldots, B_n$, one may simply compute Green[T, ⟨B_1, …, B_n⟩].
- *Properties of Analytic Algebras*: They are used for the confluence proof; see the explanation below.

The Green's evaluator needs just two *options*: The option **BoundaryPoints** specifies the interval on which the BVP is to be solved, the default being the unit interval. In case one wants to see a nullspace projector, a right inverse or a Green's operator in its "raw form" (meaning in the latter case that one will in general not be able to read off the Green's function), one may deactivate the option **ReduceAfterwards**, which is of course activated by default.

```
Options[GreenEvaluator]
{BoundaryPoints → {0, 1}, ReduceAfterwards → True}
?? BoundaryPoints
☑ Theorema`Evaluators`UserEvaluators`GreenEvaluator`BoundaryPoints
?? ReduceAfterwards
☑ Theorema`Evaluators`UserEvaluators`GreenEvaluator`ReduceAfterwards
```

The Green's evaluator is also central place for providing the special *parsing and formatting* used throughout the Green's suite. All the special notations are provided both on the input side (parsing) and on the output side (formatting). The former is done by the *Mathematica* function **MakeExpression**, the latter by the *Mathematica* function **MakeBoxes**. The details about these processes are irrelevant; let us just give the correspondence between the concrete and abstract syntax:

- *Multiplication Operator*: An expression of the form $\lceil f \rceil$ is represented internally as ™MultiplicationOperator[f].
- *Basis Expansion*: An expression of the form $f^{\#}$ is represented internally as ™BasisExpansion[f].
- *Boundary Action*: Expressions of the form $f^{\leftarrow}$ and $f^{\rightarrow}$ are represented internally as ™LeftBoundaryValue[f] and ™RightBoundaryValue[f], respectively.
- *Integral and Cointegral Action*: Expressions of the form $\int^{*} f$ and $\int_{*} f$ are represented internally as ™IndefiniteIntegral[f] and ™IndefiniteCoIntegral[f], respectively. The precedence of these operators is adjusted according to Convention 21.
- *Definite Integral Action*: An expressions of the form $\oint f$ is represented internally as ™DefiniteIntegral[f]. The precedence of this operator is adjusted according to Convention 21.
- *Right Inverse*: An expressions of the form $T^{\blacklozenge}$ is represented internally as ™DifferentialOperatorRightInverse[T].

Having now finished the description of the essential features provided by the Green's evaluator, let us briefly describe the main interface of the *confluence tools*, the function **ProveConfluence** in the Green's evaluator. As explained above, this function is basically a specialized shell around the

function **SPolynomials** sketched in Section 3. Since it is supposed to prove the confluence of just one system—the Green's identities hardwired in the Green's evaluator—there is no need for any arguments. Hence one can ask for the confluence proof by simply saying:

```
ProveConfluence[]
```

The main *output* from this function is of course: YES or NO, i.e. whether or not all S-polynomials have reduced to zero. This is shown at the very end of its output, together with the timing information, the total number of S-polynomials and the number of nonzero ones (which is hopefully 0). Besides this cumulative result, it gives the trace for every S-polynomial reduction (where the caption is now "The rules … and … yield the S-polynomial …:" rather than "We compute:"—see Section 3); see Computation 29 for several full examples. The justifications of the equalities linking the reduction chains are of a dual nature: They either come from a polynomial equation of Input 14 or from one of the properties of analytic algebras. In case of problems, the offending S-polynomials and the rules where they come from are repeated at the end of the computation (which was very helpful when tuning the Green's identities and the axioms of analytic algebra).

Fortunately, this last feature of **ProveConfluence** was not used in the final proof of Theorem 29…

3 Implementation Notes

Whereas the last chapter was oriented towards *users* of the Green's suite who may not be interested in any internal issues of implementation, the present chapter wants to address the *programmers*. Our goal is to give a rough outline of how the various functions described in the previous chapter have been realized and to point out some of the subtler points hidden in the code. Therefore it seems most natural to follow the same structure as in the user chapter, commenting first on some general issues of implementation, then specifically on the noncommutative polynomial reductor, the matrix evaluator, and the Green's evaluator.

The *structure* of this chapter is essentially as that of the previous one. In Section 1, we briefly discuss some general issues concerning programming in *Theorema*. Section 2 explains the architecture of the Green's suite from the programmer's point of view. As in the previous chapter, the remaining sections describe the three main components of the Green's suite: the polynomial reductor in Section 3, the matrix evaluator in Section 4, and finally the Green's evaluator in Section 5.

3.1 General Design Principles in *Theorema* Programming

In its current version, *Theorema* uses *Mathematica* for three different—and in principle orthogonal—purposes:

- Its front-end is used for convenient input and output of *mathematical formulae in natural syntax*. There is hardly any other system in the world that allows so much freedom in programming all details of parsing and formatting, covering a great portion of the notations really used by the working mathematician. (Note, however, that *Mathematica* does not—yet—officially support custom-tailored lexical analysis, which would be important e.g. for choosing a now text-like symbol like $\diamond$ to be used as an infix operator.)

- The vast *library of mathematical functions* provided by the *Mathematica* kernel as well as by its add-on packages may be used within proofs and computation if desired. We have made use of this facility for providing the built-ins of the Green's evaluator; see Section 5 of Chapter 2. (As explained in Section 1 of Chapter 2, this cannot introduce any inconsistency, because nothing is imported by default; the user always has to demand external knowledge explicitly.)

- Finally, *Mathematica* is also the implementation platform of *Theorema*. This is no final decision, because in essence we use only a few very basic rewrite functions that can be implemented in other languages like C or Java or ML. The advantage of *Mathematica* is that it is a very convenient system for rapid prototyping; its disadvantage is that it is very slow (for some tasks the slow-down is as much as 10^3).

In the present chapter, the only *Mathematica* aspect important to us is the third. Let us therefore first state some of the primary design principles of *Mathematica* programming as we see it in *Theorema*. One important aspect is that *Mathematica* provides some fairly efficient mechanisms for structuring moderately large amounts of code. In *Theorema*, we have approximately 150 different files, each containing hundreds of lines of code. The basic unit for structuring this mass is the *Mathematica* concept of packages:

- Every file corresponds to one package, having its own namespace that is typically named as the file. The namespaces—called *contexts* in Mathematica—are hierarchically organized, and their structure is typically isomorphic to the directory structure of the physical files underlying the packages.
- A package has a clear *interface* that defines which of the symbols are to be exported; all other symbols are only visible inside the package as a kind of local auxiliaries (this is even true for global variables as long as they are not exported—hence we call such variables package-global). In C++ terminology, the former symbols would be called "public" and the former "private".
- The packages are loaded on along their *dependency* graph. Each package starts by announcing its own name together with the names of those packages on which it depends. The *Mathematica* package mechanism then makes sure that everything is recursively loaded at the proper time when the top-level package is loaded; this is what happens when saying: **Needs**["*Theorema*`"].
- The system is loaded on a *dynamical* basis. When *Theorema* is first started, only the crucial packages are provided. But whenever the user issues a command mentioning certain autoload keywords like the name of user prover, the system will automatically load the necessary portion of packages for executing the command specified by the user.

The *file structure* of *Theorema* consists of the following key branches:

- The *language directory* **Theorema/Language/** contains all the material related to parsing and formatting (centered on **MakeExpression** and **MakeBoxes**, respectively), the user language (whose core functions are **Prove**, **Compute**, **Solve**), and the formal text language (providing most importantly the environments like **Theorem**, **Lemma**, **Definition**). The language directory has three subdirectories **General/**, **Syntax/**, **Semantics/**.
- The *kernel directory* **Theorema/Kernel/** containing the init file responsible for properly setting up *Theorema*. This directory also contains the file with the autoload data mentioned above.
- The *technical directories* **Theorema/General/** and **Theorema/System/** containing various functions for administering the overall data structures used by the system. The *Theorema* developer can also find some practical programs for package synthesis and debugging there.
- The *prover directory* **Theorema/Provers** contains all the basic and user provers of *Theorema*. Each of the former is typically contained in one subdirectory with a corresponding name (e.g. the PredicateProver resides in the subdirectory **PredicateLogic/**), whereas the latter ones are collected in one subdirectory named **UserProvers/**.

- The *evaluator directory* **Theorema/Evaluators** is somewhat analogous to the prover directory. However, as of now, there was essentially only one basic evaluator and no user evaluator.
- The *user directory* **Theorema/User/** containing customization files for modifying certain issues of style according to the taste of a user. This is of course only used in the local directory structure (see below).

The *Theorema* file structure as sketched above usually resides in some system-wide location, which we call the *public installation.* In addition to this installation available to everyone, a user may also create her own *private installation.* The latter has exactly the same structure as the public installation, and so *Mathematica* can use the following simple rule for resolving any package in the *Theorema* file structure. If a certain package should be loaded, the private installation is checked: if it contains a package by the name demanded, it is loaded, otherwise the analogous file of the public installation is taken. This scheme ensures a high degree of flexibility, and it provides a convenient development platform: As long as a package is not yet worked out completely, it remains in the private installation; if it is mature for common usage, it can be migrated to the public installation (which is of course layered through an appropriate versioning mechanism).

3.2 Organization of the Green's Suite

As suggested by the *Theorema* file structure explained above, we have distributed the packages of the Green's suite in a manner analogous to the user and basic provers of *Theorema*:

- The basic evaluators **ReduceNoncommutativePolynomial** and **EvaluateMatrices** reside as individual packages in the *evaluator directory* **Theorema/Evaluators/** alongside with the older basic evaluator **EvaluateStandard**.
- The user evaluator **GreenEvaluator** is located as another package within **Theorema/Evaluators/** in a *user evaluator subdirectory* named **UserEvaluators/**.
- It might seem natural to have the *notation-related code* somewhere under **Theorema/Language/Syntax/**. The basic agreement in *Theorema* was, however, to have only the universally relevant notation settings in the *language subdirectory*; all domain-specific material of this kind should be packaged together with the prover / solver / evaluator using it. Therefore we have put the corresponding code into the three packages mentioned above.

Besides the three main packages—the polynomial reductor, the matrix evaluator and the Green's evaluator—there is one more file named **GreenEnvironments** in the user evaluator subdirectory, alongside with the package **GreenEvaluator**. It contains all the *environments* that are needed by the Green's evaluator, e.g. **Formula**[*"Differential-Operator Right Inverse"*] in Input 41 or **Formula**[*"Nullspace Projector"*] in Input 43 and of course the Green's system in Input 14; all in Chapter 1. For a detailed description of how to apply these environments separately see Section 5 of Chapter 2.

The need of saving environments for later use is new in *Theorema*. Until now formal text was only saved in normal *Mathematica* notebooks, and one had to open these notebooks and evaluate the corresponding cells for recommitting the environments to the kernel. In case of the Green's evaluator, we wanted to provide a more *convenient mechanism* for providing the necessary environment knowledge to the kernel. Therefore we have implemented a function named **SaveEnviron-**

ment that accepts any environment currently known to the kernel as an argument and makes out of it a definition written to an external file such that the same environment knowledge becomes known to the kernel again as soon as the external file is loaded like a package. For example, consider the following two cells:

Formula[*"Differential-Operator Right Inverse"*, any[T],

$$T^{\blacklozenge} = \prod_{i=1,\ldots,n} \left\lceil e^{\hat{\lambda}_i\, x} \right\rceil A \left\lceil e^{-\hat{\lambda}_i\, x} \right\rceil \Bigg| \left[p = \text{poly}[T],\ n = \deg[p],\ \hat{\lambda} = \text{rad}[p] \right]$$

]

SaveEnvironment[**Formula**[*"Differential-Operator Right Inverse"*]];

When evaluating these two cells, the following will happen: As usual in evaluating environment specifications, the first cell commits the knowledge about **Formula**[*"Differential-Operator Right Inverse"*] to the kernel. Evaluating the second cell will then write a definition like this to some predefined external file:

```
Formula["Differential-Operator Right Inverse"] :=
  •fml["Differential-Operator Right Inverse", •range[•simpleRange[•var[Tma‘T]]], True, •
      flist[
        •lf["",
          ™Equal[™DifferentialOperatorRightInverse[•var[Tma‘T]],
            ™WithLocalValues[•range[
                •locval[Tma‘p, poly[•var[Tma‘T]]],
                •locval[•var[Tma‘n], deg[Tma‘p]],
                •locval[•var[OverHat[Tma‘λ]], rad[Tma‘p]]],
              ™ProductOf[•range[•integerRange[•var[Tma‘i], 1, •var[Tma‘n]]], True, ™Times[
                ™MultiplicationOperator[™Power[™E,
                    ™Times[™MatrixSubscript[•var[OverHat[Tma‘λ]], •var[Tma‘i]], Tma‘x]]], A,
                ™MultiplicationOperator[™Power[™E,
                    ™Times[™Minus[™MatrixSubscript[•var[OverHat[Tma‘λ]], •var[Tma‘i]]], Tma‘x]]]]]]]]]]
```

Evaluating this definition has the same effect as evaluating the first of the two cells displayed above. Hence the environment information will be made available to the kernel when the external file containing this definition is loaded. Now the structure of the file "GreenEnvironments.nb" is such that it contains—besides some administrative functions—*pairs of cells* like the one above, the first always being an environment specification and the second a corresponding call to **SaveEnvironment** for writing the definition to the external file, which is defined to be "GreenEnvironments.m".

The programmer of the Green's evaluator uses this file as follows: Whenever she wants to make some changes to the environments to be used by the evaluator, she modifies the corresponding specifications in the file "GreenEnvironments.nb" accordingly. Having done so, she simply evaluates the package initialization (this can be done by the menu command Kernel → Evaluation → Evaluate Initialization). This will *automatically create* a new version of "GreenEnvironments.m", which is the source file used by the Green's evaluator.

3.3 Implementation of the Polynomial Reductor

Before going into any details about the implementation, we should answer the following fundamental question: *Why* should one create a new evaluator in the first place? In principle, one could use the *Theorema* all-purpose evaluator **EvaluateStandard** also for reducing polynomials. This would be very cumbersome, though, for the following reasons:

- Handling *normalization rules* such as polynomial expansion should be internalized both for efficiency and clarity. Adjoining rules like $(a + b)\, c \to ac + bc$ to the reduction rules coming from polynomial equalities would cause a considerable chaos and slowdown of the overall computation. Moreover, several axioms such as additive commutativity cannot be handled in a purely rewrite fashion.

- As explained in Section 3 of Chapter 2, we want to use a multi-level strategy that breaks the reduction process into several logical units that he have called *reduction phases*. Such a strategy is not supported by **EvaluateStandard**.

- We want to be able to see the *complete trace information* about a reduction, including all the redex occurrences operated on in a particular step. The tracing support of **EvaluateStandard** shows only the equality that was used in a particular step—which is precious little information in long polynomials as those considered in the Green's evaluator.

Having thus established the necessity of a dedicated evaluator for reducing noncommutative polynomials, let us now raise one question concerning a fundamental design decision. In *Mathematica*, rewriting can in principle be done by two different mechanisms: *transformation rules or downvalues with attributes* (see the *Mathematica* book for details about these issues). The latter method has the obvious advantage of normally being more efficient, but one loses some of the finer points of tracing and evaluation control. In a previous version of the evaluator, we have provided an option for deciding between traced computations via transformation rules and untraced ones via downvalues / attributes. We have eliminated this choice in the current version, using only transformation rules for the following reasons:

- The straight-forward implementation with downvalues / attributes turned out to be *slower* than its counterpart with transformation rules! This was probably due to the attributes used for associativity and commutativity, since they waste a lot of time in AC matching, whereas the transformation rules simply keep the polynomials in flat and ordered form throughout.

- When using the attributes **Orderless** for enforcing commutativity of addition, it is not possible to specify any particular *term ordering*.

- In most cases, we wanted to have *trace information* such that any supposed speedup by downvalues / attributes becomes negligible anyway.

Though not as natural as the traditional combination of downvalues with attributes, it might also be worthwhile to try out an implementation that is fairly similar to the present one but uses *conditional downvalues without any attributes* for effecting the transformations now governed by rules. We have not yet explored this approach, but the gain in speed would probably not be substantial.

Let us now describe the *overall structure* of the computational process carried out by the reductor. As explained in Section 3 of Chapter 2, we use an approach combining normalization and reduction steps, which are governed by the following main loop:

- The input term is first fed through a postprocessor ensuring that we have a *polynomial term*, meaning an expression generated by the following grammar:

  PolTerm:: = PolTerm + PolTerm | PolTerm ∗ PolTerm | Atom

Atom:: = Indeterminate | Coefficient

Thus we have to eliminate all symbols foreign to the signature, in particular subtraction ($a - b$ is rewritten into $a + (-1)\, b$ with -1 regarded as a complex number) and powers are either resolved into iterated multiplications or—in case they are actually pseudopowers (see Section 3 of Chapter 2)—into the appropriate indeterminates. Note that other symbols outside the signature just specified are understood as unknown complex numbers.

- The polynomial term is then *normalized* according to the usual axioms of noncommutative polynomial rings: First of all, domain simplification according to the given built-ins is carried out. For example, a polynomial like $X_{4/2}$ might be simplified to X_2, where $X_{\square}$ is an indeterminate. Second, the polynomial term is expanded and its numerical coefficients are extracted and aggregated in front of the resulting monomial forms. This creates expressions described by the following grammar (using blank notation for sequence variables—see the *Mathematica* book):

  PolyForm:: = ™Plus[Monomial___]

  MonForm:: = ™Times[NumCoeff, Atoms___]

  Here NumCoeff is a complex number constant and Atoms___ stands for a sequence of atoms as specified before, except that it must not contain complex number constants. If there is no explicit numerical coefficient in a monomial form, NumCoeff is of course set to 1. Note also that the number of monomial forms may be zero (giving the polynomial form 0), as well as the number of atoms within a monomial (giving the monomial form 1). Third, like monomial forms within such a polynomial form are collected. For example, $2\,X + Y + 7\,X$ is rewritten to $9\,X + Y$. Fourth, the monomial forms are ordered within the polynomial form according to the term ordering specified by the user. This finished the normalization of polynomial terms, and the resulting normal forms will then be polynomial forms in the stricter sense introduced in Section 4 of the Appendix.

- The polynomial form thus generated is now subjected to *reduction*. If none of the given polynomial equalities are applicable, we have a canonical form with respect to the reduction system, and we can go to the next step. Otherwise, we apply the next best reduction rule that is available in the current phase (which is changed in another loop on top of the main loop described here). The resulting polynomial term will usually not be in normal form anymore, hence we go back to the previous step.

- Finally, we pass the canonical form produced by the reduction process to a *postprocessor*, which rewrites the canonical polynomial form "in a nice way". For example, we would normally want to see $1\,X + 1\,Y$ as $X + Y$ and the empty addition as 0. Moreover, we reintroduce the minus symbol at this point, rewriting a polynomial form like $X + (-Y)$ to $X - Y$. Iterated multiplications are also transformed back into powers now, e.g. producing $X^2\,Y + 3\,Y^3$ from $1\,X\,X\,Y + 3\,Y\,Y\,Y$.

Before describing some more detailed implementation issues, let us clarify the steps occurring in the *normalization process* a bit further. We start from the identities characterizing the variety of noncommutative unital rings:

$$a + (b + c) = (a + b) + c \tag{43}$$

$$a + 0 = a \tag{44}$$

$$a - a = 0 \tag{45}$$

$$a + b = b + a \tag{46}$$

$$a\,(b\,c) = (a\,b)\,c \tag{47}$$

$$a\,(b + c) = a\,b + a\,c \quad \wedge \quad (b + c)\,a = b\,a + c\,a \tag{48}$$

$$1\,a = a \tag{49}$$

Since we assume that $\mathbb{C}$ is contained in the coefficient ring, the symbols 0, 1, $-$ of the signature become superfluous, and Equalities (44, 45, 49) are absorbed into the computational laws for complex numbers, combined with the above-described conventions for polynomial forms. Equalities (43, 47) are already taken care of by using the flat symbols **™Plus** and **™Times**. Moreover, Equality (46) is already covered as well, because we order monomial forms within a polynomial form according to a user-defined term ordering as mentioned above. Hence only Equality (48) remains to be treated in normalization: It is of course applied from left to right, and this is exactly what we have called by the popular term *polynomial expansion*.

The *interface* to the polynomial reductor package is the function **ReduceNoncommutativePolynomial**. It carries out the following steps:

- The *options* passed to it are processed in the usual *Theorema* fashion, assigning their values to package-global variables.
- The reduction process is *initialized* by clearing the *Theorema* computational storage object and setting up recognizer functions for each indeterminate specified by the user.
- Then the given *knowledge bases*—the visible, hidden and built-in part—are processed, building up a suitable representation of the given polynomial equalities, including those corresponding to the option **Units**.
- The reduction system contained in this representation is now *learned* by constructing a pool of transformation rules for each reduction phase, assigning them to a package-global variable that is indexed by the strings denoting the phase labels.
- Finally, the *computation* is carried out in the way explained above. This is done by the crucial function named **DoReduction**, which we will describe in some more detail below.
- The *result* of the computation is then displayed together with the *tracing information* unless the latter was suppressed.

The main function for *executing the computation* is **DoReduction**, and it proceeds essentially as follows:

- The given expression is first *preprocessed* and echoed in the trace.
- It is then *normalized* and passed to the recursive function **DoAllPhases**; its result is then postprocessed, echoed to the trace, and returned to the main interface.

- The function **DoAllPhases** *loops over the list of reduction phases*, using the *Mathematica* function **Fold**. For each phase, the function **DoPhase** is applied to the current polynomial form and the corresponding phase label.
- Now the phase-looping function **DoPhase** lets the *pool of transformation rules associated with the given phase* operate on the input polynomial form until it stabilizes; this is done by the *Mathematica* function **FixedPoint**. In fact, the operation applied at each step is not just replacement with respect to the current pool or rules but rather a more comprehensive function **DoRuleAndUbiquitous**.
- Naturally, the function **DoRuleAndUbiquitous** does *two things*: First it applies the function **DoRule** with the current phase label, and second the function **DoUbiquitous**.
- The function **DoRule** is the *actual engine* that does the replacement with respect to the pool of rules associated with the given phase label. It uses the *Mathematica* function **Replace** for this purpose. The result of the rule application is normalized such that all the intermediate expressions are ensured to be legitimate polynomial forms.
- The function **DoUbiquitous** is somewhat analogous to **DoRule**, only that it uses the pool of rules associated with the ubiquitous "phase" (which is actually represented as a phase labeled by the empty string), and it does not only one replacement but a whole saturation cycle. This is effected by operating the *Mathematica* function **FixedPoint** on the function **DoRule**, instantiated by the ubiquitous rules.

The *code* implementing this scheme is quite readable and short, so we list it here for illustration purposes.

```
Clear[DoReduction, DoAllPhases, DoPhase, DoRuleAndUbiquitous, DoRule, DoUbiquitous];
DoReduction[poly_] :=
    Module[{input, result, output},
      TraceReduction[PostprocessPolynomial@PreprocessPolynomial@poly, "…"];
      input = NormalizePolynomial@PreprocessPolynomial[poly];
      result = DoAllPhases[input];
      output = PostprocessPolynomial[result];
      TraceReduction[output];
      output
    ];
DoAllPhases[poly_] :=
    Fold[DoPhase, poly // DoUbiquitous, $ReductionPhases];
DoPhase[poly_, phase_String] :=
    FixedPoint[DoRuleAndUbiquitous[#, phase] &, poly];
DoRuleAndUbiquitous[poly_, phase_String] :=
    DoRule[poly, phase] // DoUbiquitous;
DoRule[poly_, phase_String: ""] :=
    Replace[poly, $RulesOfPhase[phase]] // NormalizePolynomial;
DoUbiquitous[poly_] :=
    FixedPoint[DoRule, poly];
```

The *tracing functionality* does not show up in this code because it is already built into the transformation rules. This can be seen by looking at their internal representation. The example below shows the rule that is generated from the equality "DA" in the Green's system. (Here we have entered the private context of the reductor in order to avoid long context paths cluttering the essential structure of the rule.)

```
Begin["Theorema`Evaluators`ReduceNoncommutativePolynomial`Private`"];
```

```
$RulesOfPhase["1. Equalities for Isolating Differential Operators"][[1]] // InputForm

™Plus[presum___, ™Times[prefac___, ™D, A, postfac___],
 postsum___] :> (TraceReduction[PostprocessPolynomial[
   ™Plus[presum, ™Times[prefac, FramedExpression[
     ™Times[™D, A]], postfac], postsum]], "DA";
 ™Plus[presum, ™Times[prefac, 1, postfac], postsum])
End[];
```

As we can see, this rule applies to anything having outermost symbol **™Plus** (which is the case for all polynomial forms) and containing within it an expression headed by **™Times** (all summands of a polynomial form are of course monomial forms and hence meet this condition), which must have at least two factors **™D** and ***A*** right next to each other (note that ™D is the internal form of the *Theorema* symbol *D*, thus protecting it from the *Mathematica* interpretation of total derivatives). Any such polynomial form is then replaced by an expression of the form (TraceReduction[…]; ™Plus[…]), obviously consisting of *two parts*.

The first part of this expression has the side effect of recording the *trace information* in some special-purpose data structure, whereas its result with respect to the replacement is discarded as indicated by the semicolon in *Mathematica*. The second part is just the original polynomial except that it has only one factor 1 instead of the two factors **™D**, ***A***; so its effect is to *replace D A* within a polynomial by 1. Observe that the original polynomial is written to the trace object, framing the two factors **™D** and ***A***, thus signifying the redex in the trace. Moreover, this polynomial is also fed through the postprocessor for making it look nice.

One could also ask here why we do not simply generate a rule of the type

```
™Times[prefac___, ™D, A, postfac___] :→ ™Times[prefac, 1, postfac]
```

instead of the more circumstantial

```
™Plus[presum___, ™Times[prefac___, ™D, A, postfac___], postsum___] :→
  ™Plus[presum, ™Times[prefac, 1, postfac], postsum]
```

encountered in the example above. The reason is that it is much more efficient to take advantage of our knowledge about the context of the redex. This allows us to use the *Mathematica* function **Replace** rather than the usual **ReplaceAll**, which is usually denoted by the famous Mathematica **/;** or slash-semi command. Whereas applying the former to an arbitrary expression **expr** means "replace **expr** by …", the latter means "replace any subexpression of **expr** by …". Clearly the second task is much more complex and is therefore slower in *Mathematica*. That is why we prefer to use **Replace**, as one can see in the code of **DoReduction** reproduced above.

Finally let us say something about the implementation of the function **SPolynomials** used in the *confluence tools*. Its setup is completely analogous to the evaluator interface **ReduceNoncommutativePolynomial**, only that it executes

```
SPoly /@ CriticalWords[visiblekb]
```

instead of the function **DoReduction**. The function **CriticalWords** does what one expects: For each combination of two equalities in the visible knowledge base **visiblekb** (including combinations an equality with itself), it checks whether they *overlap* and aggregates a list of triples for all those that do. These triples contain the name of the first rule, the name of the second rule and the overlapping monomial form. For example, the monomial forms *D A* and *A D* would give the overlapping monomial form *D A D*, and reversing roles would give the overlapping monomial form *A D A*.

The function **SPoly** mapped over the list of triples generated by **CriticalWords** will of course create the *S-polynomial* associated with each triple. For example, if we have the triple {"DA",

"AD", $D\,A\,D$} associated with the first overlap example mentioned above, the function **SPoly** will use the rules "DA" and "AD" on $D\,A\,D$, yielding $1\,D$ and $D(1-L)$, respectively. Then it forms their difference, which is $1\,D - D(1-L)$ in our case. Finally, this polynomial is put into standard form, giving $D\,L$ in this example.

Using the rule "DL", this S-polynomial will of course be reduced to zero!

3.4 Implementation of the Matrix Evaluator

The matrix evaluator is implemented on top of the default evaluator **EvaluateStandard**. Its *outermost loop* consists of the *Mathematica* construct **FixedPoint** used to the following operation: First apply the default evaluator to the current expression, adopting all the given knowledge bases and the options. Then use a certain pool of transformation rules, called the matrix built-ins, on them until saturation is achieved; this is done with the *Mathematica* **//.** construct. Finally, another transformation rule, called the contraction built-in, is used just once; so this time the *Mathematica* **/.** construct is used for accomplishing the transformations.

The *matrix built-ins* are in effect just implementations of all the matrix operations listed in Section 4 of Chapter 2. Operations are only carried out if the involved matrices are given *in concreto*. For example, the transformation rule

```
™Plus[matrices___™Matrix] :→ (MapThread[™Plus, MatrixToList /@ {matrices}, 2] // ListToMatrix)
```

applies only to a sum of concretely given matrices (since they are always wrapped by the **™Matrix** tag). This is the case when using a constructor of the form

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

explained in Section 4 of Chapter 2 and whenever a matrix is derived from such concrete matrices. Note that we have to use some *pre- and postprocessing functions* called **MatrixToList** and **ListToMatrix**, respectively. This is necessary so that we can draw on the list functions of *Mathematica*, namely **MapThread** in the example above. The pre- and postprocessors mediate between our internal representation

```
™Matrix[™Tuple[™Tuple[a, b], ™Tuple[c, d]]]
```

and the plain *Mathematica* lists

```
{{a, b}, {c, d}},
```

all exemplified by the matrix given above. The purpose of having our own internal representation is of course to keep control over parsing, evaluating and formatting according to our own will.

The case of *multiplication* is a bit more subtle. We have to distinguish multiplication of matrix by matrix, scalar by matrix, and matrix by scalar (the last two cases obviously do not occur for addition—at least we do not consider them). Accordingly, we have three transformation rules in the matrix built-ins:

```
™Times[pre___, a_?ScalarQ, A_™Matrix, post___] :→
  ™Times[pre, Map[™Times[a, #] &, A, {3}], post]
™Times[pre___, A_™Matrix, a_?ScalarQ, post___] :→
  ™Times[pre, Map[™Times[#, a] &, A, {3}], post]
™Times[pre___, A_™Matrix, B_™Matrix, post___] :→
  ™Times[pre, Inner[™Times, MatrixToList[A], MatrixToList[B], ™Plus] // ListToMatrix, post]
```

The *first two rules* use the classifier **ScalarQ** for recognizing scalars; it uses the type inference mentioned in Section 4 of Chapter 2. They are implemented via the *Mathematica* function **Map**, which "throws" the scalar multiplication onto each entry of the matrix. Since we use the internal representation with nested tuples and **™Matrix** wrapped around (see above), these entries live at level 3 of the expression tree; this is the meaning of the last argument of the **Map** function. Note that we do not have to change the internal representation into lists since the **Map** function can deal with expressions having arbitrary head symbols.

The *third rule* is again based on the head symbol **™Matrix** restricting application to two concretely given matrices occurring next to each other within in a product. The matrix product is implemented using the *Mathematica* function **Inner**, which allows us to introduce the *Theorema* versions **™Times** and **™Plus** for the multiplication and addition of entries, respectively. The function **Inner** expects plain *Mathematica* lists, so this time we have to use the pre- and postprocessors again.

The *contraction built-in* consists of the sole transformation rule

```
™Matrix[™Tuple[™Tuple[a_]]] :→ a
```

used for replacing 1×1 matrices by the single entry they contain.

There are numerous *other operations* that one could (and should) implement in a matrix evaluator. But matrix algebra is clearly not a focal point of interest of the material to be developed here. We have therefore restricted ourselves to the absolute minimum needed for our purpose of computing Green's operators.

3.5 Implementation of the Green's Evaluator

The Green's evaluator is implemented as a cascade consisting of the following *three stages*:

- The first module applied to the input expression is named **UnfoldDefinitions**. This is essentially a call to the default evaluator **EvaluateStandard** using the *relevant definitions* for the nullspace projector, Wronski operator, differential-operator right inverse and Green's operator (see Section 5 of Chapter 2 for details) as well as some built-ins for executing the operations wron, left, right, poly, deg, rad (again we refer to Section 5 of Chapter 2 for details) and for expanding product quantifiers.

- The second module applied thereafter is named **SimplifyMatrices**, and it is basically just a cover of the *matrix evaluator* **EvaluateMatrices**, calling it with no other knowledge than the action operators listed in Section 5 of Chapter 2. The reason for including the action operators already at this point is that applying them here will usually result in some considerable simplifications in the result of this module. Of course, other simplifications will only be possible in conjunction with the reduction process effected by the next module.

- The third module finally applied is named **SimplifyPolynomial**, as it starts the *polynomial reductor* on the result of the previous module. The reductor is of course properly supplied with the necessary options, e.g. specifying the indeterminates via

```
Indeterminates → {A, B, D, L, R, ⌈□⌉}
```

  The reduction phases are set as discussed in Section 2 of Chapter 1, and trace generation is suppressed. The Green's system is specified as the visible knowledge base (unless the

option **ReduceAfterwards** has been deactivated), and the built-ins are again the action operators as above plus the function effecting basis expansion.

Let us now briefly discuss the *built-ins* implemented in the Green's evaluator package.

Basis expansion is accomplished by **BasisExpansion**, which is a recursively defined function supposed to expand all multiplication operators occurring in a polynomial form. Its crucial clause is the following:

```
BasisExpansion[⌈™Plus[args__]⌉] :=
    BasisExpansion[⌈#⌉] & /@ ™Plus[args];
```

In its first instance, this causes the expansion of a given polynomial form to split into expansions of each of its monomial forms. But more importantly, this will also resolve a multiplication operator like $\lceil x^2 + x^3 \rceil$ into $\lceil x^2 \rceil + \lceil x^3 \rceil$, thus effecting the additive breakdown into basis elements.

Similarly, constant factors are extracted as follows:

```
BasisExpansion[⌈™Times[pre___, λ_?ConstantQ, post___]⌉] :=
    ™Times[λ, ⌈™Times[pre, post]⌉ // BasisExpansion];
```

This will transform a multiplication operator like $\lceil 2\, x^2 \rceil$ to $2 \lceil x^2 \rceil$, thus effecting the homogeneous breakdown into basis elements. The other definition clauses of this function are essentially special cases of the two clauses above (dealing e.g. with products and sums having only one argument) and some clauses for closing the recursion.

The subsidiary function **ConstantQ** used above for *checking constancy* is needed for avoiding expansions like $\lceil x\, x^2 \rceil$ transforming to $x \lceil x^2 \rceil$. It evaluates to false iff the argument contains an x or a **•var** or an integration operator. The check for **•var** is necessary for the confluence proof, because there we have to deal with multiplication operators $\lceil f \rceil$ that are induced by an unknown function f; the internal form of such function variables is **•var[f]**.

The *action operators* are implemented by appealing to *Mathematica* in a straight-forward way. The actions for the left and right boundary action, differentiation, indefinite integral and cointegral are realized by the following definitions:

```
LeftBoundaryValue[term_] :=
  MathematicaSimplify[term /. Symbol["x"] → $BoundaryPoints[[1]]]
RightBoundaryValue[term_] :=
    MathematicaSimplify[term /. Symbol["x"] → $BoundaryPoints[[2]]];
Derivative1[term_] :=
    Mma2Tma[System`D[Tma2Mma[term], Symbol["x"]]];
IndefiniteIntegral[term_] :=
    Mma2Tma[Integrate[Tma2Mma[term], {Symbol["x"], $BoundaryPoints[[1]], Symbol["x"]}]];
IndefiniteCointegral[term_] :=
    Mma2Tma[Integrate[Tma2Mma[term], {Symbol["x"], Symbol["x"], $BoundaryPoints[[2]]}]];
```

The only tricky point about these functions is that they do not refer directly to the symbol x denoting the independent variable of functional expressions. The symbol x is instead referenced by **Symbol["x"]**. The reason for this slightly unusual way of addressing symbols is that the above definitions should be applicable within a *Theorema* computation. Such computations are always carried out in the protected context **Theorema`Computation`** in order to ensure an unpolluted namespace, thus avoiding any unwarranted external knowledge. Any symbols occurring in the input are then transferred into this context; in particular, the symbol x occurring in the multiplication operators of polynomials will be translated into **Theorema`Computation`x**. If we had simply used x in the above definitions, *Mathematica* would have understood this as **Tma`x**, because $x$ lives in the *Theorema* standard context **Tma`;** consequently the definitions would not apply to the input polynomials. This problem is now avoided by using **Symbol["x"]**, because this expression

denotes the symbol x in whatever context it currently occurs—hence yielding **Theorema`-Computation`x** at the time of carrying out the computation. (Obviously we could have used **Theorema`Computation`x** directly instead of the slightly cryptic **Symbol["x"]**, but this would fail as soon as the name of the computational context of *Theorema* changes.)

The *polynomial operations* poly, deg, poly are realized by the functions **CharacteristicPolynomialOfDifferentialOperator**, **DegreeOfPolynomial**, **RootsOfPolynomial**, respectively. The former simply packs the coefficients of the given differential operator into a *Theorema* tuple, as this turns out to be a convenient polynomial representation for the purposes needed here. Consequently, the function **DegreeOfPolynomial** has the trivial implementation given by the *Mathematica* function **Length** (minus one, to be precise) used for counting the number of entries in a list (which may of course also have the head **™Tuple**). Finally, the function **RootsOfPolynomial** uses the *Mathematica* command **Solve** for obtaining the roots of the given polynomial by solving the corresponding polynomial equation; the result is then packed into a row matrix using the **™Matrix** format common in the matrix evaluator.

The operation implementing the *Wronski matrix* denoted by wron is the function **WronskiMatrixOfDifferentialOperator**. It first constructs the fundamental system using an auxiliary function **FundamentalSystemOfDifferentialOperator**, then applies the *Mathematica* function **D** for differentiating each of its entries up to $n-1$ times, where n is the number of fundamental solutions; the result is of course represented using the **™Matrix** format again.

The auxiliary function **FundamentalSystemOfDifferentialOperator** calls the *Mathematica* function **DSolve**, which guarantees to solve any linear differential equation with constant coefficients. The result returned by this command is one function containing n integration constants, whose names may be chose by the option **DSolveConstants**. We use a local variable **C** for this name, thus obtaining a generic solution with n parameters named $C[1]$, …, $C[n]$. In order to obtain the corresponding *fundamental system*, we create a table whose i-th entry is the generic solution with $C[1]$, …, $C[i-1]$, $C[i]$, $C[i+1]$, …, $C[n]$ replaced by 0, …, 0, 1, 0, …, $C[n]$, with i running from 1 to n.

The functions left and right used for extracting the *left and right boundary matrix* for a given system of boundary operators are implemented by **LeftBoundaryMatrixOfBoundaryOperators** and **RightBoundaryMatrixOfBoundaryOperators**, respectively. They are realized by some trivial *Mathematica* book-keeping commands that shuffle the entries around as needed.

Finally, let us say a few words about the implementation of the top interface of the *confluence tools*, namely the function **ProveConfluence**. As explained in Section 5 of Chapter 2, this function is basically a shell built around **SPolynomials**. The latter is called with the Green's system just as in the main function **GreenEvaluator**, but it does not get any built-ins except for basis expansion (note that we still need transformations like $\lceil 2f \rceil$ expanding to $2\lceil f \rceil$, where f is of course a function variable here). The resulting S-polynomials are then reduced by an auxiliary function named **ReduceSPolynomial**, and some logging information is displayed (see Section 5 of Chapter 2 for details).

The function **ReduceSPolynomial** is basically a call to the polynomial reductor **ReduceNoncommutativePolynomial**, again using most settings like in the main function **GreenEvaluator**. The crucial difference is that the properties of analytic algebras as specified in Theorem 28 of Chapter 1 are given as an additional argument, and again only the rules for basis expansion are used as built-ins.

And the main result of this theorem was that **ReduceSPolynomial** returned always the same polynomial: zero!

Appendix: The Concept of Polynomial

Polynomials of some particular kind—not the 'usual' commutative polynomials—will be the main building material for the methods presented in this thesis. Therefore we will now take the time for explaining the *concept of polynomial* in rather great detail and placing it into its proper mathematical context. For this purpose, we will follow a very frequent historical pattern: The first attempt of defining polynomials in Section 1 seems confusing and worthless, rather a kind of silly mythology. Therefore, it is replaced by a succinct and rigorous definition in Section 2, which does away with all those mysteries and provides a solid basis for effective computer implementation. After some deeper considerations, though, one is led back to the seemingly mystic old definition, now in the clear light of modern logic and structure theory; this is what we will show in Section 3. This provides not only a new appreciation of the old approach but even an alternative way of implementation that may include many other types of polynomials as expounded in Section 4. Finally we will discuss in Section 5 how to "adjust" the degree of noncommutativity to the level; this is relevant in Chapter 1.

A.1 A Sloppy Definition

Polynomial computation lies at the heart of algebra and, in particular, computer algebra. Some people have even gone so far as to characterize computer algebra as the art of polynomial manipulation. Now we would not subscribe to this view as a reasonable *definition* of computer algebra (we think that the approach in [44] is rather satisfactory; see also page 2 in [70]). We rather understand this dictum as an *observation* expressing the ubiquity of polynomials throughout computer algebra (note also the title of [70], which is the coursebook for computer algebra at our institute). Beyond any doubt, the polynomials belong to the *core notions* of this discipline, deeply penetrating its theoretical foundations as well as the practical machinery of its algorithms. While some people might hesitate to call it *the* most fundamental concept of computer algebra, it is at least fair to say that it shares this role with a few mates like matrix, ideal or algebraic extension.

Given their paramount importance, polynomials ought to be introduced with conceptual clarity and impeccable rigor. Therefore it is a bit surprising that conventional textbooks of mathematics are often vague and sloppy when they come to polynomials. In [54] on page 95, the usual polynomials over a unital and commutative ring R are introduced thus (we have slightly adapted the words to our present setting): "The elements of an algebraic ring extension $R[\alpha]$ can obviously be obtained from the *formal expressions* $c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$ with $c_0, \ldots, c_n \in R$ and $n \in \mathbb{N}$ if one replaces x by the element α. We call

$$f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \tag{50}$$

a *polynomial* in x over R; the symbol x is called its indeterminate." Similar 'definitions' can be found in numerous introductory textbooks of mathematics.

What is the problem with the above 'definition'? First of all, the name "formal expression" is rather *mysterious*. We will resolve the mystery soon, but for now let us assume that we are naive and innocent students of mathematics. So we assume that we know what an "expression" like $c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$ means; after all, we have dealt with such expressions numerous times. Now let us look at the *logical structure* of the whole statement. The 'variables' $R, n, c_0, \ldots, c_n$ are to be quantified universally such that we can instantiate them for concrete

examples like $2 + x$, arising from the instantiation $R \leftarrow \mathbb{Z}$, $n \leftarrow 1$, $c_0 \leftarrow 2$, $c_1 \leftarrow 1$. Strictly speaking, this is already problematic: Under the quantifier, we have here a flexible sequence of 'variables' that expands magically to the number of variables needed for each case. But first-order predicate logic does not allow such magic (but see for example [41] for a new paradigm that does allow this 'magic' in a completely rigorous setting). By considering $c_0, \ldots, c_n$ as variables, we are dealing with indivisible symbols just like $A, B, C, \ldots$, so their indices are purely typographic embellishments like the serifs in "A"—and this is certainly not the intention of the author. What he really wants to say is, "for any natural number n and any R-valued sequence c having a support of length n" (the support of a function is that part of its domain where the function maps into a number different from zero). He could even abbreviate this by saying, "for any R-valued sequence c with finite support"; we will come back to this point presently. In any case, a term like c_n is now a compound of the intended kind: the sequence (function with domain $\mathbb{N}$) c applied to the natural number n. Understood in this way, equation (50) can be seen as an abbreviation for

$$\mathrm{f}(\mathrm{x}) = \sum_{\mathrm{i}=0}^{\mathrm{n}} \mathrm{c_i}\, \mathrm{x}^{\mathrm{i}}, \tag{51}$$

and the sum quantifier appearing here can be introduced in the usual way (see for example page 58 in [14]). All this could be seen as a meticulous expansion of the above sentence into solemn predicate logic, as it is usually expected from the 'mature mathematical reader'. Still we contend that formulating the statement in the right way does not take any additional effort, and it makes life a bit easier (not only for the not-so-mature reader).

So we see that the variables R, n, c are universally quantified, but what about the 'variable' x, which is mystically called an *indeterminate*? It is here that we run into real trouble. On the one hand, x should be *quantified universally* such that we can substitute any 'number' $a \in R$ in (51) for in order to evaluate $f(a)$. On the other hand, we want to regard a polynomial like $2 + x$ as a definite expression; as such, it cannot depend on the value of some 'global variable' x: If this variable is set to some fixed number a, all the 'polynomials' are nothing else than numbers, many of them coinciding (depending on the choice of a)! For example, take $2 + x$ as above and set a to 0; now it coincides with $2 - x$ and $2 + x^2$ and $2 + 3\,x - 7\,x^2$, etc. This is clearly not what we mean by polynomials! Hence the global variable cannot have a fixed value, which means that x should be *quantified existentially*. But how can we quantify a variable both universally and existentially?!?

There is another problem with the above 'definition'. Consider a typical polynomial computation like

$$(2 + \mathrm{x})\,(-2 + \mathrm{x}) = -4 + 2\,\mathrm{x} - 2\,\mathrm{x} + \mathrm{x}^2 = -4 + \mathrm{x}^2. \tag{52}$$

What is the intermediate term in this equality chain? Is it a polynomial? Obviously not, since we cannot have two distinct linear coefficients c_1. But if it is not a polynomial, what is it? A mysterious creature that dissolves itself as soon as the 'real' polynomial appears?

A.2 A Rigorous Definition

Fortunately, the textbooks of computer algebra usually do not indulge in such mysteries. They introduce the polynomials over a coefficient ring R, denoted by $R[x]$, as the *ring of all R-valued sequences with finite support*, endowed with appropriate operations for addition, subtraction, and multiplication (see for example page 672 in [29] or page 17 in [70], where one can find detailed definitions of the concepts touched in this section). We have already come across these sequences when we analyzed the logical structure of the mystic 'definition' given above. The new definition as we find it in virtually all modern textbooks continues this line of thought to its logical end: The sequence c in (51) is really the only essential ingredient of a polynomial; all the rest is mystic accessories!

The definition via sequences is very elegant and terse; we will therefore from now on call it the succinct definition. One can also paraphrase it as follows: The ring of polynomials $R[x]$ is constructed as $R \oplus R \oplus \ldots = R^{(\mathbb{N})}$, the *direct sum* of countably many copies of R, and their multiplication is given by the Cauchy product of sequences. At this point, we cannot resist the temptation of mentioning the ring of formal power series, typically denoted by $R[[x]]$: Dropping the finiteness condition on the sequences, one arrives at $R \times R \times \ldots = R^{\mathbb{N}}$, the *direct product* of countably many copies of R. Here we encounter the same phenomenon as with the polynomials—the succinct definition was able to dispel an infamous mystery long associated with power series: How is it possible that one can derive so many useful identities, even when the involved power series are divergent? An extreme example is given on page 347 in [32], which is concerned with solving a recurrence equation. As usually, the procedure starts by encoding the required sequence c in the coefficients of a suitable power series $C \in \mathbb{R}^{\mathbb{N}}$ such that we must now solve for C. After some manipulation, one ends up with the differential equation

$$C'(x) = x^2\, C''(x) + 3\, x\, C'(x) + C(x) \tag{53}$$

and initial conditions $C(0) = C'(0) = 1$. Using hypergeometric series techniques, one can solve (53), arriving at

$$C(x) = \sum_{n=0}^{\infty} n!\, x^n. \tag{54}$$

From this power series, one can immediately read off the sequence c solving the original recurrence equation, namely $c_n = n!$. Admittedly, this is not a very impressing problem. But it does make the point addressed before: The power series C is as divergent as it can be—namely in the whole complex plane except for the origin! Still it was very useful in solving the recurrence equation… Such a situation seemed quite mysterious to the earlier mathematicians, hence they called this notion "formal power series", meaning that they simply *ignored convergence questions*; see page 206 in [32]. The succinct definition clarifies this issue completely—ignoring convergence means that we are actually not dealing with power series but with plain sequences, together with some operations that are constructed to imitate those on the corresponding power series: Besides the ring operations, one can also use differentiation as in (54), integration, division, composition, etc. So polynomials and power series are merely a convenient language frame for talking about sequences, and sometimes they behave similar to their functional counterparts. So the only left-over mystery is maybe this: Is it just coincidence that everything fits together so nicely? We will find an interesting answer to this question in Section 3.

Of course, everything generalizes smoothly to *multivariate polynomials*. The only difference is that we have to consider multisequences instead of sequences, i. e. functions $\mathbb{N}^n \to R$ instead of $\mathbb{N} \to R$. So the ring of multivariate polynomials and formal power series, accordingly denoted by $R[x_1, \ldots, x_n]$ and $R[[x_1, \ldots, x_n]]$, has the carriers $R^{(\mathbb{N}^n)}$ and $R^{\mathbb{N}^n}$, respectively; the operations are defined analogously. For details, see the literature cited above.

It seems that we have now clarified the mysteries addressed in Section 1. So let us try to answer the two main questions that came up there. The first one is: What is the mystic *indeterminate*? Let us first consider the univariate case $R[x]$. In terms of the succinct definition, x is nothing else than the sequence $\langle 0, 1 \rangle$. Within the ring $R[x]$ or $R[[x]]$, one cannot see much special in the polynomial x; only the corresponding compositional structure reveals it as its neutral element. Passing to a multivariate polynomial ring $R[x_1, \ldots, x_n]$, the real meaning of the indeterminates $x_1, \ldots, x_n$ becomes clear when we realize that every *poly*nomial splits into a sum of *mono*mials, and every monomial consists of a coefficient in R and a *power product*. The latter form a monoid, often denoted by $[x_1, \ldots, x_n]$, and in this monoid, the indeterminates serve as the *primitive generators*. So the in-determinates turned out to be very determinate polynomials within the ring! This also concludes an earlier line of thought: In the logical analysis following (51), we made the statement that an indeterminate must be either existentially quantified (essentially a constant) or universally quantified (essentially a free variable). Now we see that the first option is true—indeterminates are object constants for denoting the 'basis' of $[x_1, \ldots, x_n]$.

Having clarified the nature of the indeterminates, we are led to another question associated with them: How does evaluation work now? Since a polynomial like $-4 + x^2$ is interpreted as a sequence $\langle -4, 0, 1 \rangle$, we cannot just substitute a number like 3 for x. After all, x is not a variable but just another polynomial! Hence we must interpret evaluation at a number $a \in R$ as another operation on polynomials, which we will denote by eval_a. For example, we have $\mathrm{eval}_3(-4 + x^2) = 5$. The operation eval_a is known as *evaluation homomorphism*, since it turns out to respect the polynomial ring structure; see for example page 147 in [53]. Analogously, there is an evaluation homomorphism eval_a at $a \in R^n$ for a multivariate polynomial ring $R[x_1, \ldots, x_n]$.

With the evaluation homomorphism available, we can also clarify the idea of the "functional counterparts" mentioned above: We can associate with each polynomial $p \in R[x_1, \ldots, x_n]$ a function $\tilde{p} : R^n \to R$ defined by $\tilde{p}(a) = \mathrm{eval}_a(p)$; this $\tilde{p}$ is known as the *polynomial function* associated with (or: induced by) p. We denote the ring of all n-ary polynomial functions over R by $\mathcal{P}^n(R)$; it is a subring of the ring R^R of all n-ary functions on R. At the first glance, one might think that the structures $R[x_1, \ldots, x_n]$ and $P^n(R)$ are more or less the same, but this is an illusion: The mapping $p \mapsto \tilde{p}$ is a ring epimorphism, which means that the polynomials are much more fine-grained than the polynomial functions. Take, for example, $R \leftarrow \mathbb{Z}_2$. Then polynomials like $x + x^2$ will all collapse into zero functions. In fact, there are only four distinct functions in $P^2(\mathbb{Z}_2)$, while there are infinitely many polynomials (as for any coefficient ring).

Hence one can understand the polynomial ring as an *algebraic model of polynomial functions*, taking into account only the ring operations (functional addition, zero function, negative function, functional multiplication, one function) and ignoring all the 'accidental' features arising from evaluation in the particular coefficient ring. The power of the polynomial concept comes from the fact that many important 'patterns' become clearer by fading out accidental details of this kind. Going a bit further in this direction, we could get a first suspicion why the indeterminates might really be in-determinate: A polynomial in $\mathbb{Z}_2[x]$ like $x + x^2$ above would not collapse into the zero function if we think of the x as coming from a 'generic' ring (we will later call such a ring "free"); such a generic x is like a Joker card—the only thing we know about it is that it lives in *some* ring. Seen in this way, it really deserves the name "indeterminate". For more about this issue, we must again refer to Section 3.

Finally let us also answer the second question asked at the end of Section 1: What is this *interme diate term* in the equality chain (52)? Obviously, we cannot call it a polynomial according to the succinct definition. All we could say is that it is the sum of four polynomials, all of which happen to be monomials, but this is not completely satisfying if one is honest. The point is that the definition of the polynomial operations yield 'finished' polynomials at once. While this may be of advantage in some situations where we do not want to deal with any 'computational details', one could also ask for a more fine-grained polynomial concept that attaches an appropriate meaning also to such "intermediate terms". The succinct definition clearly does not satisfy this desire; the intermediate terms remain 'ghosts'. It is therefore high time to move on to the new approach which we have already advertised so strongly.

A.3 An Alternative and More General Definition

Before starting this new approach, let us cast a glance at the *literature*. A comprehensive treatment of the material be found in [42]; see especially pages 1-40. A lucid summary of this rather lengthy treatment is given in [16], where things are analyzed from the viewpoint of symbolic computation. The vast majority of computer algebra texts, however, are restricted to what we have called the succinct definition. Even [50] (note the subtitle!), a recent monograph on polynomials, does not hint at any alternative definition. For doing full justice to [54], we should also mention that on page 94 they provide a correct version of the succinct definition along with the rather suspicious 'definition' criticized above. (One could actually get the impression that they feel guilty about the vague description given on page 95, so they try to compensate by adding the succinct definition. If they had not declared the vague description as their 'official' definition, one could interpret the latter as intuitive ideas that are to be formalized in the succinct definition, albeit on a slightly different path.)

One may wonder why the succinct definition is so fashionable in our days. The reason for this is very simple: As we will see later, it is somewhat *superior in computation* when compared to the alternative definition to be discussed now. Besides this, the old definition requires a considerably greater technical apparatus that does not pay off as long as one studies only 'ordinary' polynomials. As we will see later, the succinct definition is an optimized specialization of the 'general' polynomials, obtained by custom-tailoring the case of the ordinary ones.

As announced at the beginning of this chapter, we will have to consider some of those general polynomials, in particular several kinds of non-commutative polynomials. It is possible to find custom-tailored representations for them similar to the succinct definition of ordinary polynomials, such an approach is rather ad-hoc. The alternative definition provides an *elegant conceptual frame* for a uniform description of all those general polynomials—including the ordinary commutative ones as well as the 'extraordinary' non-commutative ones. Furthermore, it establishes a suitable basis for discussing possible custom-tailored representations.

In the preceding section, we have already encountered some ideas that support the case of the good old indeterminate. So it should not come as a surprise that the announced alternative approach is *not really new*—we will henceforth call it the "old definition"—but in need of careful formal treatment for putting certain delicate intuitions on a rigorous logical basis.

Let us start by reconsidering the original ideas presented in connection with the 'definition' (50): The polynomials are regarded as "formal expressions", built up from an "indeterminate", from numbers, and from the operations $+$, $-$, $*$ (or other operations if we consider general polynomials). One could respond to this: "*Any* mathematical theory consists of more or less formal expression, often containing variables and various other operations like sin and $\int$. So why do they

mention this fact for the polynomials?" But the difference is that the polynomials are regarded as *formal* expressions. We will make this clear immediately.

As explained above, one may regard the polynomials as functions that have forgotten their evaluation in the coefficient ring. So we have to construct objects that 'behave like' polynomial functions over a certain coefficient ring, except for evaluation. Now we come to a very crucial point, which we will first express in terms of daily life: For communicating various ideas about politics, philosophy or ethics, we can point at numerous constellations in real life. When it comes to very subtle and dense ideas, though, it may be necessary for the expert communicators—called playwrights—to invent some constellations—called dramas—that could also appear in principle (we are not talking about absurd drama and the like). When the piece is put on stage, people can point at the constellations depicted there as if they occurred in real life; in fact, the events as such *are* real life. Without intending any philosophical implications, we will transfer this principle to mathematics now: Having the idea of "polynomial functions that have forgotten their evaluation", we can either search for some known mathematical objects exhibiting these properties (in case of the ordinary polynomials, the finite sequences of the succinct definition would serve this purpose) or we can *construct appropriate objects by staging their properties*. This is what is meant by *formal* expressions: they are put on stage.

In mathematics, dramaturgy is studied in model theory and universal algebra; the dramas are the canonical models and Herbrand models. Before we turn to look at these literary genres a bit more closely, let us briefly meditate the overall significance of the theatrical viewpoint. Again we take daily life as our guiding principle: Drama belongs to the fine arts, and some people would even consider it as the highest form of art. Artists contend that their art comprises, in principle, all of life. In particular, a (sufficiently powerful) dramatist should be able to depict any aspect of life, be it ever so subtle. In mathematics, the situation is again similar: In 1930, Kurt Gödel proved(!) that 'everything can be put on stage', provided that it is 'consistent with reality'; see [31]. Reflecting objects of immediate experience creates new objects that can in turn be experienced on a higher level; this ability of reflection appears to be a fundamental function of the human mind, penetrating the arts as well as mathematics. B. Buchberger calls the reflection step the *transition to the meta level*, and he considers it to be the essence of logic and the driving force of mathematical development; see [15]. In the *Theorema* group, we will therefore invest some effort into providing computer support for this key step of mathematical creativity in the near future.

Seeing this central role of model construction—in our metaphor: the art of staging—it will certainly pay off to consider the introduction of polynomials in the clear light of the general situation. We will do this by reviewing a modern proof of Kurt Gödel's model existence theorem mentioned above: Every consistent set of formulae has a model. From this one can easily infer that every formula that is a consequence of an axiom system is also deducible from the axioms by the proof calculus; this is known as the *Completeness Theorem*. The converse statement that every formula deducible from an axiom system is also a consequence of the axioms is also true and is considerably easier to prove; it is known as the Soundness Theorem. Combining completeness and soundness, one sees that the semantic consequence relation and the syntactical deduction relation actually coincide, which shows the adequacy of the deduction system with respect to the intended meaning. For the precise formulation and proof, we refer to [28] and [24]. For our purposes, it will be sufficient to sketch the proof of the model existence theorem. We will first recollect some basic notions dealing with models.

We are given a set of consistent formulae, which we will call the *axiom system* Φ, and we want to construct a model for this axiom system. In general, we expect Φ to have many models (the special case of categorical axiom systems, i. e. those having a unique model, is rather rare). Typically, the collection of all models is so big that it forms a proper class. In the literature, it is called

the *model class* of the axiom system and denoted by Mod(Φ); see for example page 108 in [24]. Following B. Buchberger [8], we prefer to call this collection the *category* described by Φ and its elements as the corresponding *domains* or *structures* (if there are no predicate symbols, the domains are also called *algebras*). In the sequel, we will explain his most important ideas of formalizing such notions; we will try to realize most of them in the *Theorema* system during the next years.

Although it is not our intention to move to the *viewpoint of category theory*, we will point out certain connections here and there (see [47] and [25]). First of all, it is clear that model classes are indeed categories in the sense of category theory. (Since we are thinking in terms of the ordinary set-theoretic semantics, the model classes are actually concrete categories, meaning that they are founded on the category of sets. The viewpoint of category theory would be to abstract away from the "arbitrariness" of the set category.) Their arrows are naturally given by the homomorphisms in the sense of model theory; see page 225 in [24]. Of course, one may prefer to use other arrows in some situations (for example taking continuous functions instead of isometries in the category of metric spaces), and the salient feature of category theory is that it gives much greater importance to the arrows of a category rather than to its objects. Since we will not be working in category theory proper, our emphasis will be different—we regard categories as elementary building blocks for characterizing mathematical objects.

As an example that will later become important for constructing the ordinary polynomials, we consider the case when Φ consists of the axioms for unital commutative rings; this yields the *category of unital commutative rings* (with the ring homomorphisms as their natural arrows). In *Theorema*, one way of describing this category is by giving the definition (essentially coming from [8], see also page 71 in [64])

$$
\begin{aligned}
&R : \mathit{UniCommRing} \iff \\
&\Bigg(\underset{R}{+} : \underset{R}{\diamond} \times \underset{R}{\diamond} \to \underset{R}{\diamond} \bigwedge \underset{R}{0} : \underset{R}{\diamond} \bigwedge \\
&\quad \underset{R}{-} : \underset{R}{\diamond} \to \underset{R}{\diamond} \bigwedge \underset{R}{*} : \underset{R}{\diamond} \times \underset{R}{\diamond} \to \underset{R}{\diamond} \bigwedge \underset{R}{1} : \underset{R}{\diamond} \bigwedge \\
&\quad \underset{x,y,z:\underset{R}{\diamond}}{\forall} \left(x \underset{R}{+} y\right) \underset{R}{+} z = x \underset{R}{+} \left(y \underset{R}{+} z\right) \bigwedge \underset{x:\underset{R}{\diamond}}{\forall} x \underset{R}{+} 0 \underset{R}{=} x \bigwedge \\
&\quad \underset{x:\underset{R}{\diamond}}{\forall} x \underset{R}{+} \left(\underset{R}{-} x\right) = 0 \underset{R}{=} \bigwedge \underset{x,y,z:\underset{R}{\diamond}}{\forall} x \underset{R}{+} y = y \underset{R}{+} x \bigwedge \\
&\quad \underset{x,y,z:\underset{R}{\diamond}}{\forall} \left(x \underset{R}{*} y\right) \underset{R}{*} z = \left(x \underset{R}{*} y\right) \underset{R}{*} z \bigwedge \underset{x:\underset{R}{\diamond}}{\forall} 1 \underset{R}{*} x \underset{R}{=} x \bigwedge \\
&\quad \underset{x,y,z:\underset{R}{\diamond}}{\forall} x \underset{R}{*} y = y \underset{R}{*} x \bigwedge \\
&\quad \underset{x,y,z:\underset{R}{\diamond}}{\forall} x \underset{R}{*} \left(y \underset{R}{+} z\right) = x \underset{R}{*} y \underset{R}{+} x \underset{R}{*} z \Bigg).
\end{aligned}
\tag{55}
$$

First of all, note that we have used the *typing predicate* ":" for describing that *R* lies in the class of all unital commutative rings, denoted by *UniCommRing*. We could as well use a unary predicate

"IsUniCommRing", writing IsUniCommRing(R) instead of $R : \mathit{UniCommRing}$, but the usage of the binary typing predicate permits a more uniform treatment (one may view this notion of type as in a sorted logic). The important point is that UniCommRing is a proper class, so we cannot say $R \in \mathit{UniCommRing}$, but the predicate ":" is a sufficient substitute since we do not need any nesting. We call *UniCommRing* the type of category of unital commutative rings, and we will use italics for denoting such category types. Informally, category types are usually identified with the actual categories, so *UniCommRing* "is" the category of unital rings.

The typing convention is also used for the domain predicate: For example, $x : \underset{R}{\diamond}$ means that x lies in the carrier of R. In this case, we could use $x \in \underset{R}{\diamond}$ instead, but using the domain predicate allows us to overload operators when we regard the domain predicate as an appropriate type (see below). Furthermore, we note the *curried operator symbols*: For example, $\underset{R}{*}$ is just notation for $R(*)$, so that $x \underset{R}{*} y$ stands for $R(*)(x, y)$. We follow the tradition of model theory in bundling the available operations into a so-called *operation object* which identifies the particular domain; in our case, the operation object is R.

Finally, let us mention that one could easily make the above definition even more readable by realizing a couple of common *syntactic conventions* (which we will presuppose from now on): First, the operation object is *confused with the carrier* (even in heterogeneous structures with more than one carrier—like vector spaces—one can distinguish a particular one—like the vectors), i. e. whenever R is on the right-hand side of "$\in$", it is replaced by $\underset{R}{\diamond}$. Second, the operator symbols are *overloaded*, e. g. when we encounter $x * y$ with x and y both having the type "lying in the carrier", it is replaced by $x \underset{R}{*} y$. In the type declarations of the operator symbols, we may anyway leave out the underscript without danger of confusion. Third, we can indicate that all variables in the definition scope are *relativized* to the carrier by a certain external declaration (in the heterogeneous case, one may introduce finer-grained declarations—for example denoting vectors by Latin letters and scalars by Greek ones). Fourth, we will write the operation object *for non-operator symbols as a subscript* rather than an underscript. Fifth, one may use juxtaposition for denoting multiplication. Using these conventions, the above definitions reads

$$
\begin{aligned}
&\mathrm{R} : \mathit{UniCommRing} \iff \\
&\quad (\ + : \mathrm{R} \times \mathrm{R} \to \mathrm{R} \ \wedge\ 0_{\mathrm{R}} : \mathrm{R} \ \wedge\ - : \mathrm{R} \to \mathrm{R} \ \wedge\ * : \mathrm{R} \times \mathrm{R} \to \mathrm{R} \ \wedge\ 1_{\mathrm{R}} : \mathrm{R} \ \wedge \\
&\qquad (\mathrm{x} + \mathrm{y}) + \mathrm{z} = \mathrm{x} + (\mathrm{y} + \mathrm{z}) \ \wedge \\
&\qquad \mathrm{x} + 0_{\mathrm{R}} = \mathrm{x} \ \wedge\ \mathrm{x} + (-\mathrm{x}) = 0_{\mathrm{R}} \ \wedge\ \mathrm{x} + \mathrm{y} = \mathrm{y} + \mathrm{x} \ \wedge \\
&\qquad (\mathrm{x}\,\mathrm{y})\,\mathrm{z} = (\mathrm{x}\,\mathrm{y})\,\mathrm{z} \ \wedge\ 1_{\mathrm{R}}\,\mathrm{x} = \mathrm{x} \ \wedge\ \mathrm{x}\,\mathrm{y} = \mathrm{y}\,\mathrm{x} \ \wedge \\
&\qquad \mathrm{x}\,(\mathrm{y} + \mathrm{z}) = \mathrm{x}\,\mathrm{y} + \mathrm{x}\,\mathrm{z}\ ).
\end{aligned}
\tag{56}
$$

In an extensive 'formal library', one would certainly *split such a definition* using a suitable system of hereditary subdefinitions (and we plan to do this in the *Theorema* formalization project) describing e. g. semigroups, monoids, groups, commutativity, unitality, distributivity; this will not be necessary for our present purposes. Besides this, the type declarations of the first line (often formulated as "closure properties" like additional axioms: the sum of two ring elements is again a ring element) is naturally separated from the remaining ones. We call the first line the *signature axioms* of the domain, the remaining lines its *proper axioms*. This separation will also be important for what follows, so let us do it in detail: We agree that we will provide a signature predicate for each category, which we will verbalize as "like" similar to Mizar e. g. in "group-like" ; see [65]. For example, the signature predicate for the above category would say that R "is like a unital

commutative ring" (for categories with a shorter name like groups this sounds better: the corresponding domains would be called "group-like"). We will denote this predicate by

$$\begin{aligned} &R : [\mathit{UniCommRing}\,] \Longleftrightarrow \\ &\quad (+ : R\times R \to R \ \wedge\ 0_R : R \ \wedge\ - : R \to R \ \wedge\ * : R\times R \to R \ \wedge\ 1_R : R\ ). \end{aligned} \tag{57}$$

Furthermore, we will assume that every category definition presupposes the corresponding signature definition implicitly (otherwise is can hardly make sense). Hence we can now write the definition in (56) as

$$\begin{aligned} &R : \mathit{UniCommRing} \Longleftrightarrow \\ &\quad ((x+y)+z = x+(y+z) \ \wedge\ x+0_R = x \ \wedge\ x+(-x) = 0_R \ \wedge\ x+y = y+x \ / \\ &\qquad (x\,y)\,z = (x\,y)\,z \ \wedge\ 1_R\,x = x \ \wedge\ x\,y = y\,x \ \wedge \\ &\qquad x\,(y+z) = x\,y + x\,z\ ). \end{aligned} \tag{58}$$

Let us now go back to the proof of the model existence theorem: We are given such a consistent axiom system Φ like the formulae contained in the right-hand side of (58). They describe a certain category like that of the unital commutative rings in the above example. Our task is to prove that this category contains at least one domain, i. e. we have to construct one particular domain out of the information provided in Φ. As explained before, the crucial idea is to put the axioms 'on stage'. So the first task is to clarify the 'cast of the actors', i. e. the carrier of the desired model. Since we should be able to 'see' every object that we can possibly talk about, the most natural choice is to take the set of all closed terms as a carrier (since we can always reduce the given axioms to closed formulae); a function symbol works on such a closed term in the natural way by concatenation. The resulting structure is called the *ground term algebra* $\mathcal{T}_\Sigma$ induced by the signature of Σ, which is determined by the function and constant symbols occurring in the axioms.

For example, if Φ contains the axioms of unital commutative rings

$$\begin{aligned} &(x+y)+z = x+(y+z), \\ &x+0 = x, \\ &x+(-x) = 0, \\ &x+y = y+x, \\ &(x\,y)\,z = (x\,y)\,z, \\ &1\,x = x, \\ &x\,y = y\,x, \\ &x\,(y+z) = x\,y + x\,z, \end{aligned} \tag{59}$$

the signature is given by

$$+ : R\times R \to R \ \wedge\ 0 : R \ \wedge\ - : R \to R \ \wedge\ * : R\times R \to R \ \wedge\ 1 : R\ . \tag{60}$$

We see that (59) is very similar to (58), and (60) is very similar to (57), but we should also understand the difference: Now the axioms (59) are universally quantified over the whole universe R, and the operation symbols are not bundled into any operation object (model itself is the bundle); the signature (60) is therefore not part of the axioms but a syntactic declaration on the meta level. Since we do not consider heterogeneous structures (which could be treated analogously in a sorted logic), the signature information boils down to *specifying the arity of each function symbol* (object constants are regarded as nullary function constants). Hence we can regard Σ as the (set-theoretic) function; in the example above it would be

$$\Sigma = \{\langle +, 2\rangle, \langle 0, 0\rangle, \langle -, 1\rangle, \langle *, 2\rangle, \langle 1, 0\rangle\}. \tag{61}$$

Note that we can obtain the set of function symbols in use by taking the domain $\mathrm{dom}(\Sigma)$ of the signature; for example, we have $\mathrm{dom}(\Sigma) = \{+, 0, -, *, 1\}$ in the above example. Given a Σ of this form, we can always go back to the *corresponding signature definition* over some domain D, namely

$$\bigwedge_{f\in \mathrm{dom}(\Sigma)} f : D^{\Sigma(f)} \to D, \tag{62}$$

which we will denote by $\mathrm{Sig}_\Sigma(D)$. Note the meta-level universal quantifier in this formula: upon substituting a concrete Σ and D, the formula (62) is supposed to end up in a conjunction of the form (60). In the current presentation, we could actually be pedantic in using various meta-level symbols for distinguishing them from the corresponding object-level symbols, e. g. the equality symbol or the set braces. However, as the context usually eliminates these ambiguities, we will suppress this distinction for the sake of simple notation.

The carrier set of the desired term algebra over Σ is supposed to consist of all closed terms *and nothing else*. This is an inductive set, which we will call the set of *words* over Σ and denote by W_Σ. This is a generalization of the natural numbers, which can be regarded as words over the signature $\{\langle 0, 0\rangle, \langle {}^+, 1\rangle\}$, where the superscript-plus denotes the successor function. In the case of natural numbers, the Peano axioms do achieve the inductive definition (for details see page 204 in [61]): The 'basis axioms' $0 \in \mathbb{N}$ and $\forall_{n\in\mathbb{N}}\ (n \in \mathbb{N} \Rightarrow n^+ \in \mathbb{N})$ ensure that the function symbols of the signature serve as constructors; the 'equality axioms' $\forall_{n\in\mathbb{N}}\ n^+ \neq 0$ and $\forall_{n,m\in\mathbb{N}}\ (n^+ = m^+ \Rightarrow n = m)$ banish ambiguity in the term representation; the 'induction axioms' $\Phi_{n\leftarrow 0} \wedge \forall_n\ (\Phi \Rightarrow \Phi_{n\leftarrow n^+}) \Rightarrow \forall_n\ \Phi$ for all formulae Φ (particularly those containing a free occurrence of n) ensure the "nothing else" passage. We can do something completely analogous for specifying the set of words through

$$\begin{aligned}
&\bigwedge_{f\in \mathrm{dom}(\Sigma)}\ \bigwedge_{t_1,\ldots,t_{\Sigma(f)}\in W_\Sigma} \bar{f}\,(t_1, \ldots, t_{\Sigma(F)}) \in W_\Sigma,\\
&\bigwedge_{f\in \mathrm{dom}(\Sigma)} \left(\bigwedge_{s_1,\ldots,s_{\Sigma(f)}\in W_\Sigma}\ \bigwedge_{t_1,\ldots,t_{\Sigma(f)}\in W_\Sigma} \bar{f}\,(s_1, \ldots, s_{\Sigma(f)}) = \bar{f}\,(t_1, \ldots, t_{\Sigma(f)}) \Rightarrow \bigwedge_{i=1,\ldots,\Sigma(f)} s_i = t_i \right),\\
&\bigwedge_{f\in \mathrm{dom}(\Sigma)} \left(\left(\bigwedge_{t_1,\ldots,t_{\Sigma(f)}\in W_\Sigma} \Phi \Rightarrow \Phi_{t\leftarrow \bar{f}\,(t_1,\ldots,t_{\Sigma(f)})} \right) \Rightarrow \bigwedge_t \Phi \right),
\end{aligned} \tag{63}$$

where Φ runs through all formulae (particularly those containing a free occurrence of t). Here we have to make a small but crucial remark: The function symbols of the given signature Σ have to play a double role: first, they appear as letters in the words just introduced; second, they will be used for denoting the operations to be defined on the words. We distinguish them by overbarring

the function symbols when they are used as letters. The overbar works like a *quote*: Consider the situation where $s(x)$ denotes the servant of a person x, and H denotes the Prince of Denmark. In the 'term algebra of stage play', $s(x)$ will accordingly be defined as the "servant" of an actor x (the quotes should signify that we really mean the actor who plays the servant of the character embodied by the actor x) . Then $s(\overline{H})$ is the "servant" of the actor "the Prince of Denmark", but $\bar{s}(\overline{H})$ is the actor "the servant of the Prince of Denmark". Both descriptions are of course equivalent due to our definition of s, and this is precisely why we call such a definition *natural*.

We can now construct the term algebra over Σ by providing the carrier set W_Σ with the natural operations that are induced by concatenation. For doing this, we will use what we call a *functor* in *Theorema*; see [10], [11]. Let us explain this briefly: Whereas a category describes a domain implicitly by stating certain axioms that it should fulfill, a functor defines it explicitly in terms of zero, one or more given domains. Typically, a category contains many domains, but a functor always constructs exactly one. This is analogous to the difference between relations and functions: A relation like $y \sim x$ will typically yield many objects y for a fixed x, but a function like $y = f(x)$ will always yield only one y for a fixed x. Functors that do not take any input domain are called introduction functors for obvious reasons (of course they may be parametrized like any other functor, i. e. taking several other input arguments); category theory would see them as functors from the category of sets. In the comparison to functions and relations, the introduction functors correspond to the object constants (which we have identified with the nullary function constants). In *Theorema* (see [68]), we would specify the functor for constructing the direct product of two additively written groups G and H by the following construct (indices denote tuple components here):

Definition["Direct Product of Groups", any[G, H],

CartesianProduct[G, H] = Functor[P, any[x, r, s, u, v],

$$\mathbb{s} = \langle + : P\times P \to P,\ 0 : P,\ - : P \to P\rangle$$

$$\underset{P}{\in}[x] \Leftrightarrow \left(\underset{G}{\in}[x_1] \bigwedge \underset{H}{\in}[x_2]\right)$$

$$\langle r, s\rangle \underset{P}{+} \langle u, v\rangle = \left\langle r \underset{G}{+} u,\ s \underset{H}{+} v\right\rangle$$

$$\underset{P}{0} = \left\langle \underset{G}{0},\ \underset{H}{0}\right\rangle$$

$$\overline{}^{P}\langle r, s\rangle = \left\langle \overline{}^{G} r,\ \overline{}^{H} s\right\rangle$$

]]

Logically, we can express this conveniently by using Hilbert's ι-quantifier (introduced in [36]; see also the chapter on the ι- and ε-quantifiers on page 27ff of the recent investigation [30]). Using a notation similar to that of the category predicates presented before, we have

$$\begin{aligned} \mathrm{G}\otimes\mathrm{H} = \underset{\mathrm{P}}{\iota}\Big(\underset{\mathrm{P}}{+} : \mathrm{P}\times\mathrm{P}\to\mathrm{P} \bigwedge 0_{\mathrm{P}} : \mathrm{P} \bigwedge \overline{}^{\mathrm{P}} : \mathrm{P}\to\mathrm{P} \bigwedge \\ \mathrm{x} : \underset{\mathrm{P}}{\diamond} \Leftrightarrow \left(\mathrm{x}_1 : \underset{\mathrm{G}}{\diamond} \bigwedge \mathrm{x}_2 : \underset{\mathrm{H}}{\diamond}\right) \bigwedge \langle \mathrm{r},\ \mathrm{s}\rangle \underset{\mathrm{P}}{+} \langle \mathrm{u},\ \mathrm{v}\rangle = \left\langle \mathrm{r} \underset{\mathrm{G}}{+} \mathrm{u},\ \mathrm{s} \underset{\mathrm{H}}{+} \mathrm{v}\right\rangle \bigwedge \\ 0_{\mathrm{P}} = \langle 0_{\mathrm{G}},\ 0_{\mathrm{H}}\rangle \bigwedge \overline{}^{\mathrm{P}}\langle \mathrm{r},\ \mathrm{s}\rangle = \left\langle \overline{}^{\mathrm{G}}\mathrm{r},\ \overline{}^{\mathrm{H}}\mathrm{s}\right\rangle\Big). \end{aligned} \tag{64}$$

This means that the direct product of the groups G and H is "such a group P that its addition is a binary operation, its neutral element a nullary operation, its inverse a unary operations; such that its carrier predicate is determined by the carrier predicates of the factor groups; and such that the

three operations are defined componentwise as indicated". Since we will not use *Theorema* for manipulating category and functor definitions, we will stick to the traditional notation (64) rather than the *Theorema* notation (63). Using this notation and assuming the obvious definitions, we can now state theorems like

$$\mathrm{G} : \mathrm{Group} \wedge \mathrm{H} : \mathrm{Group} \implies \mathrm{G} \otimes \mathrm{H} : \mathrm{Group}. \tag{65}$$

Observe also the philosophy of avoiding domain questions on the input arguments to functors: In the definition (64), we have not required G or H to be a group, not even a semigroup or a groupoid; let us call such properties *domain restrictions*. The definitional also 'works' if we supply a domain without these properties; of course, the resulting domain $G \otimes H$ will not have these properties either. All that we are saying in (65) is that *if* both G and H are indeed groups, *then* so is $G \otimes H$. The input arguments G and H cannot be 'completely messed up' (meaning that they do not provide the appropriate signature), though, as long as we regard the binary copula predicate ":" in it as a type declaration in the sense of Higher Order Logic; let us call these sanity conditions the *typing restrictions* of the functor. Of course, in some borderline cases the distinction between domain and typing restrictions may be blurred; for example, positivity could be a domain restriction expressed by the predicate $x > 0$ or a typing restriction expressed by the declaration $x : \mathbb{R}^+$. In most cases, however, there is a very natural difference: The typing restrictions should be kept 'easy' since they are to be done at 'compile time' (the question of whether a term is well-typed belongs to syntax checking just as excluding formulae like $\varphi \wedge \wedge \psi$), whereas the domain restrictions may be arbitrarily complex like the execution of a program at 'run time' (in general, the task of checking domain restrictions may become a full-blown theorem-proving job). In particular, some domain predicates are not even recursive and therefore not at all suitable for typing questions!

Let us also mention some brief remarks about the connections to category theory. Obviously, the latter has a more restricted notion of functors, namely that they must also respect the arrows in the involved categories. Let us call such functors *respectful* (also showing our respect for MacLane and Eilenberg's work). In many cases, one does indeed meet such respectful constructions; the direct product of groups in (64) is a case in point.

Coming back to the construction of the term algebra $\mathcal{T}_\Sigma$, we can finally specify it by an appropriate introduction functor. We have already defined the set of words W_Σ as its carrier. Since we are reusing each function symbol in Σ as an 'actor' that imitates its action on the desired term algebra T, the latter can simply take over the signature prescribed in Σ, giving the signature axioms $\mathrm{Sig}_\Sigma(T)$. So the only thing left to do is to specify the natural operations on the closed terms by concatenation, thus completing the *definition of the term algebra*

$$\mathcal{T}_\Sigma = \underset{\mathrm{T}}{\iota}\Bigg(\mathrm{Sig}_\Sigma(\mathrm{T}) \bigwedge \mathrm{t} : \underset{\mathrm{T}}{\diamondsuit} \Leftrightarrow \mathrm{t} \in \mathrm{W}_\Sigma \bigwedge \underset{\mathrm{f}\in\mathrm{dom}(\Sigma)}{\forall} \underset{\mathrm{t}_1,\ldots,\mathrm{t}_{\Sigma(\mathrm{f})}\in\mathrm{W}_\Sigma}{\forall} \mathrm{f}_\mathrm{T}(\mathrm{t}_1, \ldots, \mathrm{t}_{\Sigma(\mathrm{f})}) = \bar{\mathrm{f}}(\mathrm{t}_1, \ldots, \mathrm{t}_{\Sigma(\mathrm{f})})\Bigg). \tag{66}$$

Before dealing with the predicate symbols, we add one final remark about the term algebra. Up to now, we have only considered ground terms, but it is almost trivial to extend this to terms containing any *variables* we like. Suppose we are given a set of variables X, which must of course be disjoint from $\mathrm{dom}(\Sigma)$. Then we can describe the corresponding term algebra *for* Σ *in* X, denoted by $\mathcal{T}_\Sigma(X)$, simply by regarding the variables in Ξ as new object constants—the only difference is that we do not include them in the signature of the resulting term algebra. It is a typical feature of mathematical structures that one abstracts from the practical necessity of having constructors for denoting the elements. For example, the real numbers $\mathbb{R}$ are described as a field; hence, their

signature provides only the constants 0 and 1, and the corresponding ground terms denote nothing more than the rational numbers. Therefore one usually adjoins a suitable name system (see below), but the name constants are not regarded as a part of the algebraic signature.

The identification of variables and "new" constants is commonplace in proving: "In order to prove $\forall_x \ldots$, take x arbitrary but *fixed* …" (meaning that one considers a *constant* x_0 in place of the x). Loosely speaking, the important point is not really whether we call an x a constant or a variable; the crucial question is rather whether we have any knowledge about it. Having no knowledge means that we are dealing with variables: this is why one takes an "*arbitrary* but fixed x" in the above proof situation. In the case of term algebras, we have not yet introduced any axioms, so there is no difference. In the presence of other axioms, though, we have to take care that the new constants are really *fresh* in the sense that they do not occur in the axioms. If this is the case, we have essentially added variables, which we might as well call "indeterminates"…

We will start our consideration of predicate symbols with the most fundamental of all, namely *equality*. In fact, we have already used it in (66), since any inductive structure presupposes equality. The equality predicate is in some sense special among all other predicates since it possesses the same universality as the logical connectives and quantifiers—it *always* makes sense to ask whether or not two elements are equal. Hence it is natural to stipulate that a model for a theory containing the predicate $=$ must interpret it as the 'actual' equality relation between the elements of the universe; such a model is called a *normal model*. By imposing or not imposing this semantic restriction, one can choose between treating equality 'internally' (as a logic-internal notion) or 'externally' (as all the other function and predicate symbols like $+$ or $\leq$). On the syntactic side, making equality internal corresponds to adding suitable equational inference rules or axioms to the derivability relation; one speaks of a *logic with equality* in such a case.

The external treatment of equality is straight-forward since it is the same as for any other predicate symbol; see below for how to handle them. Nevertheless, there are good reasons for considering normal models: In first-order predicate logic, one cannot characterize equality through axioms. Of course, one adds appropriate equational axioms (reflexivity and replacement axioms for each function and predicate symbol) in a logic with equality. But this cannot preclude *abnormal models*: For example, take the group axioms together with an axiom stating that every element has order three. From group theory we know that there is only one such group up to isomorphism, namely $\mathbb{Z}_3$. However, there are many non-isomorphic models containing e. g. four elements or even $\aleph_1$ elements, and we can rightly call them abnormal! Of course, all the elements are grouped into just three equivalence classes, and we cannot distinguish between their representatives through any property expressed in our language; as far as the theory is concerned, all the elements in an equivalence class are therefore 'practically equal'. Still we would prefer to have a group that 'behaves normally', and this is why we speak of normal models.

In fact, every abnormal model can be *made normal* by collapsing its equivalence classes into single elements; the normal model is simply the finest congruence on the universe. The fact that any model can be made normal is also the reason why abnormality makes no problems for reasoning: In the entailment relation, we may quantify over all models fulfilling the equality axioms (including all normal models) or over all normal models (that can be expanded into arbitrary coarser models); this makes no difference. Hence entailment for normal models is equivalent to deduction in equational logic, just as plain entailment is equivalent to derivability in plain logic (Gödel's Completeness Theorem).

Since normal models are definitely preferable and we can always obtain them, it is reasonable to construct them in the first place. In other words, we now take the option of treating equality internally in the sequel. Hence we are now in the following situation: We have to construct a

normal model for an axiom system Φ, which contains (possible after some trivial equivalence transformation on the axioms) only formulae of the form

$$\forall_{x_1} \cdots \forall_{x_n} \exists_{y_1} \cdots \exists_{y_m} \forall_{z_1} \cdots \forall_{z_k} \exists \cdots S = T,$$

where S and T are two terms having the free variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ and $z_1, \ldots, z_k$, etc. By the usual technique of Skolemization, we can get rid of the existential formulae at the expense of adding suitable function symbols to the signature. Agreeing that we abbreviate universal formulae by letting the corresponding variables occur free, we end up with axioms of the form $S = T$; we call them *equalities*. The categories that can be constructed by such axiom systems are already very rich in structure; they are called *varieties*. Their structure is studied carefully in universal algebra [20]; for example, one famous result by Birkhoff says that one can characterize varieties as those algebraic categories which are closed under subalgebras, homomorphs and quotients (see page 75). We call a category algebraic if its signature is purely algebraic (i. e. consisting of object and function constants alone).

As explained above, we can obtain a normal model by collapsing the supposedly equal elements. Hence we have to construct the term algebra for the given signature and then take the quotient with respect to the congruence relation induced by the equalities Φ. For this purpose, we need each equality $S = T$ in the form of a pair $\langle \overline{S}, \overline{T} \rangle$ containing the word corresponding to the left-hand and right-hand side (see after (63) for a discussion of the quoting function $x \mapsto \overline{x}$). Let us assume we have got a meta-level function Eql that does all the preprocessing on Φ as described above and then transforms each equality $S = T$ into the word pair $\langle \overline{S}, \overline{T} \rangle$; such a function is easy to implement but tremendously tedious to describe. Let us now look at the variety of unital commutative rings for seeing an example. Its axiom system Φ is given by the set of the following formulae (incorporating implicit universal quantifiers)

$$\begin{aligned}
&(\mathrm{x} + \mathrm{y}) + \mathrm{z} = \mathrm{x} + (\mathrm{y} + \mathrm{z}),\\
&\mathrm{x} + 0 = \mathrm{x},\\
&\mathrm{x} + (-\mathrm{x}) = 0,\\
&\mathrm{x} + \mathrm{y} = \mathrm{y} + \mathrm{x},\\
&(\mathrm{x}\,\mathrm{y})\,\mathrm{z} = (\mathrm{x}\,\mathrm{y})\,\mathrm{z},\\
&1\,\mathrm{x} = \mathrm{x},\\
&\mathrm{x}\,\mathrm{y} = \mathrm{y}\,\mathrm{x},\\
&\mathrm{x}\,(\mathrm{y} + \mathrm{z}) = \mathrm{x}\,\mathrm{y} + \mathrm{x}\,\mathrm{z},
\end{aligned} \tag{67}$$

which we have already seen in their relativized form in (58); we will soon say a little more about this relativization. For now, let us just mention that Eql_Φ is the set of the pairs

$$\begin{aligned}
&\langle \overline{\mathrm{Plus}}\,(\overline{\mathrm{Plus}}\,(\mathrm{x},\,\mathrm{y}),\,\mathrm{z}),\,\overline{\mathrm{Plus}}\,(\mathrm{x},\,\overline{\mathrm{Plus}}\,(\mathrm{y},\,\mathrm{z}))\rangle,\\
&\langle \overline{\mathrm{Plus}}\,(\mathrm{x},\,\overline{0}),\,\mathrm{x}\rangle,\\
&\langle \overline{\mathrm{Plus}}\,(\mathrm{x},\,\overline{\mathrm{Negative}}\,(\mathrm{x})),\,\overline{0}\rangle,\\
&\langle \overline{\mathrm{Plus}}\,(\mathrm{x},\,\mathrm{y}),\,\overline{\mathrm{Plus}}\,(\mathrm{y},\,\mathrm{x})\rangle,\\
&\langle \overline{\mathrm{Times}}\,(\overline{\mathrm{Times}}\,(\mathrm{x},\,\mathrm{y}),\,\mathrm{z}),\,\overline{\mathrm{Times}}\,(\mathrm{x},\,\overline{\mathrm{Times}}\,(\mathrm{y},\,\mathrm{z}))\rangle,\\
&\langle \overline{\mathrm{Times}}\,(\overline{1},\,\mathrm{x}),\,\mathrm{x}\rangle,\\
&\langle \overline{\mathrm{Times}}\,(\mathrm{x},\,\mathrm{y}),\,\overline{\mathrm{Times}}\,(\mathrm{y},\,\mathrm{x})\rangle,\\
&\langle \overline{\mathrm{Times}}\,(\mathrm{x},\,\overline{\mathrm{Plus}}\,(\mathrm{y},\,\mathrm{z})),\,\overline{\mathrm{Plus}}\,(\overline{\mathrm{Times}}\,(\mathrm{x},\,\mathrm{y}),\,\overline{\mathrm{Times}}\,(\mathrm{x},\,\mathrm{z}))\rangle,
\end{aligned} \tag{68}$$

where we have replaced the operator symbols $+, -, *$ by standard function names Plus, Negative, Times for making the overbarring more readable. But what is now the meaning of the variables x, y, z occurring there? They are now ranging over the elements in the word set W_Σ; for example, the second word pair in (68) says that for each admissible *word* x, we want to identify the words $\overline{\mathrm{Plus}}(x, \overline{0})$ and the word x itself. Therefore let us write $\mathrm{Eql}_{\Sigma,\Phi}$ for the set consisting of all such word pairs over W_Σ. For arbitrary Φ, we now define $\mathrm{Eqv}_{\Sigma,\Phi}$ to be the equivalence relation induced by the word pairs $\mathrm{Eqv}_{\Sigma,\Phi}$, i. e. the corresponding reflexive, symmetric and transitive closure. It is easy to see that $\mathrm{Eqv}_{\Sigma,\Phi}$ is actually a *congruence*, as long as Φ is consistent (this is where one needs the hypothesis in Gödel's Completeness Theorem for the case of equational axiom systems). Hence we may define all the operations of $\mathcal{T}_\Sigma$ on the congruence classes as usual. This gives rise to the functor

$$\mathcal{F}_{\Sigma,\Phi} = \underset{F}{\iota}\Big(\mathrm{Sig}_F(\Sigma) \bigwedge \; x : \underset{F}{\diamond} \Leftrightarrow x \in W_\Sigma / \mathrm{Eqv}_{\Sigma,\Phi} \; \bigwedge \; \underset{f\in \mathrm{dom}(\Sigma)}{\forall} \; \underset{t_1,\ldots,t_{\Sigma(f)}\in W_\Sigma}{\forall} \; f_F([\![t_1]\!], \ldots, [\![t_{\Sigma(f)}]\!]) = [\![\bar{f}(t_1, \ldots, t_{\Sigma(f)})]\!] \Big), \tag{69}$$

which does indeed construct the required model for the given signature Σ and the equational axiom system Φ.

For making this statement precise, we have to assert two things: that $\mathcal{F}_{\Sigma,\Phi}$ has the right syntax (it fulfills the desired signature specification associated with Σ) and the right semantics (it fulfills the equational axioms described in Φ). The difference is again the relativization incorporated in passing from the meta to the object level. We have already solved this problem for the signature by introducing the "corresponding signature definition" in (62), denoted by $\mathrm{Sig}_\Sigma(D)$; it relativizes the signature given 'absolutely' in Σ. Let us then introduce a similar meta-level function for effecting this *relativization on the axioms*; we denote it by $\mathrm{Axm}_{\Sigma,\Phi}(D)$. Again we skip the actual implementation of such a function since it is quite trivial but dull: The only thing to be done is to replace each signature constant (this is also true for the general situation which involves object, function and predicate constants) by the corresponding domain-curried symbol and to restrict all the quantifiers to the domain. Signature and Axioms make up a category as explained above, so we can combine the relativization definitions into the definition scheme

$$\begin{aligned} &D : [\mathrm{Cat}_{\Sigma,\Phi}] :\Leftrightarrow \mathrm{Sig}_\Sigma(D), \\ &D : \mathrm{Cat}_{\Sigma,\Phi} :\Leftrightarrow \mathrm{Axm}_{\Sigma,\Phi}(D). \end{aligned} \tag{70}$$

Finally, we can now formulate our result in the following convenient form: For all algebraic signatures Σ and for all equational axiom systems Φ, we have

$$\mathcal{F}_{\Sigma,\Phi} : \mathrm{Cat}_{\Sigma,\Phi}. \tag{71}$$

This result means that $\mathcal{F}_{\Sigma,\Phi}$ is indeed a normal model for the prescribed category. In logic, such a structure is called a *canonical model*; see page 202 in [28]. As for the term algebras, one may easily add a set X of "variables", thus obtaining a corresponding equivalence $\mathrm{Eqv}_{\Sigma,\Phi}(X)$ and domain $\mathcal{F}_{\Sigma,\Phi}(X)$. In universal algebra, such a structure is called the *free algebra in X for* the variety specified by Σ and Φ; see page 66f in [20], where it is also proved that free algebras for a fixed variety and fixed $|X|$ are unique up to isomorphism (this is why we can say: "the" free algebra …). In fact, their description involves an abstract semantic characterization of the model class, but they prove it equivalent with the syntactic Σ, Φ formulation used here (called "equational

classes" by them); see page 75.

Before we continue on the exciting path of Gödel's Completeness Theorem, let us briefly ask ourselves what we have achieved so far in terms of the polynomial definition. Obviously, our preferred category is that of the unital commutative rings. They are described by (55), with Σ and Φ as in (61) and (67), respectively. So what is the free algebra in the variety *UniCommRing* over the indeterminates, say, $\{x, y\}$? It is all the terms that we can build from x, y and the signature constants, subject to the equalities of the variety. For example, $\mathcal{F}_{\Sigma,\Phi}$ contains terms like $x^6\,y^2 + x^2\,y + 2\,x\,y$ or $2\,x + 2$, where we have used the power notation as an abbreviation, so $x^6\,y^2$ is actually $x\,x\,x\,x\,x\,x\,y\,y$. Furthermore, $2\,x\,y$ is an abbreviation for $x\,y + x\,y$, etc. So this *almost polynomials over the integers*, just that we do not have the actual integers available as coefficients. Besides this, we cannot simplify a term like $(1 + 1)\,x$. This is *not* the same as $x + x$, which we have abbreviated as $2\,x$. We will soon return to this point of 'installing a coefficient algebra'.

But now let us continue the development of model construction as it is needed for Gödel's Completeness Theorem, as soon as we have nonequational axioms in Φ. In fact, we will not really need this case for introducing polynomials, so we will treat this point a bit more briefly. It is just too tempting to walk the whole path to its beautiful conclusion, since there is not so much missing at this point. In fact, there is only one crucial idea necessary for coping with general formulae: One has to 'saturate' them in such a way that one can read off how to define the required relations of the model. Following [28], we find it convenient to distinguish two kinds of *saturations*:

- First one shows that it is straightforward to construct a model for Hintikka sets; page 112. Such sets are *downward saturated*, containing for each formula all the parts that are necessary for its proof; somehow one goes downward on their history. For example, if $\varphi \wedge \psi$ is in the set, then φ and ψ should be as well; if $\exists_x\,\varphi$ is in the set, then $\varphi_{x \leftarrow t}$ should also be in the set for some closed term t. The idea of the model construction is this: Going downward more and more, one must finally hit atomic formulae with ground terms. Since ground terms are just the carrier elements of the term algebra, we can define the relation symbols simply by making it true exactly for those terms that occur in the corresponding atomic formulae. (Dealing with internal equality, we must of course treat equational atomic formulae—ground equalities—as we did before, whereas everything else is done on the representatives and transferred to the congruence classes; see page 202f.)

- Second one constructs a model for a given axiom system Φ by adding *all* expressible formulae that keep Φ consistent; see page 116f. In this way, Φ becomes not only downward but also *upward saturated* as we add all the formulae that could be proved as theorems or axiomatized as independent formulae; in this sense, we are going upward in their history. This is done by using an arbitrary enumeration of all formulae (in case of uncountable languages one even has to appeal to Zorn's Lemma), so one forces completeness of the resulting axiom system Φ^* in a rather brutal manner: For an undecidable formula φ, either φ itself or its negation $\neg\,\varphi$ is adjoined—whichever happens to come first in the chosen enumeration! In this sense, the constructed model involves some arbitrariness that was not necessary as long as we had only equational axioms or Hintikka sets. This is also the reason why the corresponding term algebra in general does not obey the universal mapping property characteristic for free algebras, not even for purely algebraic axioms: Not every class of algebras has free algebras; see page 66 in [20].

Having done all this, one ends up with a *canonical model* for the given signature Σ and axiom system Φ, which we will denote by $C_{\Sigma,\Phi}$. Just as before, the completeness result now reads as $C_{\Sigma,\Phi} : \mathrm{Cat}_{\Sigma,\Phi}$. We should also mention that one speaks of *Herbrand models* if equality is treated

externally: In this case, the universe of the model is still the term algebra rather than its equational quotient, and one need not deal with representatives of congruence classes.

As a final remark on the general problem of constructing canonical models for axiomatically specified categories, let us again say a few words about the viewpoint of category theory. There, the overall tendency is always to get rid of the set category as the underlying material from which the models are built; everything is formulated in 'arrow-theoretic' terms. A good example for this approach can be found on page 147ff in [23], although this book focuses on "algebraic type theory": The signatures are purely algebraic but formulated in a typed language; the Completeness Theorem gets the name "categorical type theory correspondence". By using a suitable category-theory version of the construction sketched above, the so-called "classifying category"—correspond ing to what we have called $\mathcal{F}_{\Sigma,\Phi}$ above— for the given "algebraic theory" is constructed. Its main value is seen in being a syntax-independent representation of the algebraic theory.

Coming back to the polynomials, we must now resolve the *problem with the coefficients.* Actually, there are two problems here: one of a syntactic nature, the other of a semantic nature. The first one is that we do not even have symbols for denoting the coefficients for the desired polynomials. In the example above, we had the case of $\mathbb{Z}[x, y]$ in mind. But what about $\mathbb{Q}[x, y]$? We cannot possibly write down the term $2.5\,x + 4.7\,y$ unless we have constant symbols for all of $\mathbb{Q}$ or whatever coefficient domain we have in mind. The other problem is that we wanted to identify $(1 + 1)\,x$ and $2\,x$; this is the semantic problem now. In order to cover *all* cases, we have to add *all* the equalities valid in the desired coefficient domain.

In general, we want to define the polynomials for variety described by the signature Σ and the equational axioms Φ; see page 12 in [42]. Accordingly, the coefficient domain may be any algebra A in this variety, i. e. we are given any A with $A : \mathrm{Cat}_{\Sigma,\Phi}$. Of course, the coefficient domain may be of a much more specific type; for example, as above, it may be a field $\mathbb{Q}$ although we construct the polynomials in the variety of unital commutative rings. The specific nature of the coefficient domain enters only through the computational laws, allowing identifications like $(1 + 1)\,x$ and $2\,x$, as in the above example; this is what we meant before by describing polynomials as the model of polynomial functions. But first let us define the necessary constants for denoting the coefficients of the polynomial. Since they name each individual in A, we will call them the *name constants* for A. Let us denote them by priming the respective individual, e. g. $3'$ would be the name of 3. The *name system* of A, denoted by $\mathrm{Nm}(A)$, is then defined as $\{a' \mid a \in A\}$. For reasons of sanity, we require $\mathrm{Nm}(A)$ to be distinct from Σ, whenever a name signature is constructed (this can always be enforced if necessary). The other thing needed is the collection of all equalities $S = T$ valid in A, when S and T range over all the terms over the signature $\Sigma \bigcup \mathrm{Nm}(A)$. We will call this collection the *operation table* of A and denote it by $\mathrm{Op}(A, \Sigma)$, since this is common terminology at least for finite domains A. For uncountable domains coefficient domains like $\mathbb{R}$, such a definition of $\mathrm{Nm}(A)$ and $\mathrm{Op}(A, \Sigma)$ is clearly inconstructive, but one can usually resort to some 'reasonable' subdomain if constructivity is an issue; in the case of $\mathbb{R}$, algebraic number fields could be such a choice.

Now it is easy to describe the polynomials precisely. Let Σ be a signature and Φ an axiom system over Σ, together forming the variety $\mathrm{Cat}_{\Sigma,\Phi}$ with category type $\mathfrak{V}$. Furthermore, let A be a coefficient domain with $A : \mathfrak{V}$, and let X be a set of indeterminates, which must of course be distinct from $\Sigma \bigcup \mathrm{Nm}(A)$. Then we define the *polynomials for* $\mathfrak{V}$ *over* A *in* X as the free algebra in X for the variety with signature $\Sigma \bigcup \mathrm{Nm}(A)$ and axioms $\Phi \bigcup \mathrm{Op}(A, \Sigma)$. The corresponding introduc·tion functor for polynomials is therefore given by

$$\mathcal{P}_{\Sigma,\Phi}\,(\mathrm{A},\,\mathrm{X}) = \mathcal{F}_{\Sigma \bigcup \mathrm{Nm}\,(\mathrm{A}),\Phi \bigcup \mathrm{Op}\,(\mathrm{A})}\,(\mathrm{X}). \tag{72}$$

Having introduced a name $\mathfrak{V}$ for the category type of $\mathrm{Cat}_{\Sigma,\Phi}$, we can also write $\mathcal{P}_{\mathfrak{V}}(A, X)$ instead of $\mathcal{P}_{\Sigma,\Phi}(A, X)$. For example, we can describe the *normal polynomials* over a unital commuta

tive ring R as the domain $\mathcal{P}_{UniCommRing}(R, X)$, usually denoted by $R[X]$. In the future, we will refer to this domain simply as the "commutative polynomials" over the ring R. A bit more specifically, we might take for X the set $\{x, y, z\}$ and for R the ring $\mathbb{Z}$ or the field $\mathbb{R}$ forming the integer polynomials $\mathbb{Z}[x, y, z]$ or real polynomials $\mathbb{R}[x, y, z]$, respectively. With finite indeterminate sets like $\{x, y, z\}$, one usually speaks of "polynomials in x, y, z" instead of "polynomials in $\{x, y, z\}$", thus identifying $R[\{x, y, z\}]$ with $R[x, y, z]$.

Actually, the construction (72) is quite general—it yields what universal algebra calls a *free union*; see page 13 in [42]. Given two free algebras $\mathcal{F}_{\Sigma,\Phi}(X)$ and $\mathcal{F}_{\Sigma',\Phi'}(X')$, we can construct the free algebra $\mathcal{F}_{\Sigma\dot\cup\Sigma',\Phi\dot\cup\Phi'}\left(X \,\dot\cup\, X'\right)$, where $\dot\cup$ denotes the disjoint union (in a reasonable setting X and X' will be disjoint anyway, so $\dot\cup$ is simply the normal set-theoretic union) as their free union in the variety $\mathrm{Cat}_{\Sigma\dot\cup\Sigma',\Phi\dot\cup\Phi'}$. Strictly speaking, we should say: this is *some* free union, but by speaking of *the* free union we mean the particular construction considered here. The signatures Σ and Σ' need not be the same (the corresponding algebras are then called dissimilar), but by merging them into the (normal!) union $\Sigma \cup \Sigma'$, their common part show up only once in the resulting free union (this is actually a slight generalization of the usual definition of free union, which is restricted to similar algebras, but it seems to be a quite natural one—the notion of homomorphism in $\mathfrak{V}$ remains the same even though some of its conditions become vacuous in case of a reduced signature). About the axioms of Φ and Φ' we must of course stipulate compatibility, meaning that not only must Φ and Φ' themselves be consistent, but even their union $\Phi \cup \Phi'$.

Let us now introduce the *name algebra* $\mathcal{N}_\Sigma(A)$ of a given algebra A in a variety $\mathrm{Cat}_{\Sigma,\Phi}$ as $\mathcal{F}_{\Sigma\cup\mathrm{Nm}(A),\mathrm{Op}(A,\Sigma)}$. A typical name example would be the number system $\ldots, -2, -1, 0, 1, 2, \ldots$ for the ring of integers. But A could also be the algebra of trigonometric functions; in this case, the name algebra would include the functions $x \mapsto \sin(x)$, $x \mapsto \cos(x)$, $x \mapsto \sin(3.75\,x)$, $x \mapsto \sin(x)\cos(2\,x)$, … Now we can make the notion of "installing a coefficient algebra" precise: We see that $\mathcal{P}_{\Sigma,\Phi}(A, X)$ is indeed the free union of $\mathcal{F}_{\Sigma,\Phi}(X)$ and $\mathcal{N}_\Sigma(A)$.

As mentioned before, mathematical structures usually do not include sufficiently many constructors for denoting all the carrier elements. Hence it is customary to eliminate the name-part from the polynomial signature, retaining only the original signature Σ. In model theory, the domains arising from such an elimination are called *reducts*. As a consequence, we have the nice preservation property $\mathcal{P}_{\mathfrak{V}}(A, X) : \mathfrak{V}$. So the luxury of the name signature was just an intermediate device for defining polynomials.

Now that we have finished building up polynomials according to the old definition, let us ask ourselves how the evaluation homomorphism and polynomial functions look like in this setting. One can actually answer these questions on a very broad basis: For any free algebra, a term can be evaluated in the most natural way—simply by plugging domain values into the indeterminate. More precisely, if T is a congruence class of terms from $\mathcal{T}_{\Sigma,\Phi}$, represented by some particular term t possibly (in fact, typically) containing the indeterminate x, and a fitting algebra $A : \mathrm{Cat}_{\Sigma,\Phi}$, its *evaluation* in a carrier point $a \in \diamond_A$ is given by the value $\bar{t}\,|_{x\leftarrow a}$; we denote this by $\mathrm{eval}_{\Sigma,\Phi,A}(a, T)$ or $\mathrm{eval}_{\mathfrak{V},A}(a, T)$ if $\mathfrak{V}$ designates $\mathrm{Cat}_{\Sigma,\Phi}$. Here we have written $\bar{t}$ for the term arising from t by 'recurrying' the operation domain from $\mathcal{T}_{\Sigma,\Phi}$ to A, e. g. replacing $\mathcal{T}_{\Sigma,\Phi}(+)$ by $A(+)$. Hence $\bar{t}$ becomes a term denoting an individual of A, as soon as each occurrence of the indeterminate x is replaced by the value a; this is what we indicated by writing $\bar{t}\,|_{x\leftarrow a}$. Note that the evaluation homomorphism is well-defined by specifying its values in terms of equivalence classes, because the equivalence is a congruence.

The case of several indeterminates is analogous, but one must introduce a total ordering on them for supplying a sequence of input arguments to them (if one wants to consider infinitely many indeterminates, one actually needs a well-ordering as on page 6 in [42]). Usually one thinks of $\mathfrak{V}$ and A as fixed quantities (in programming, one would say "global variables"), so it is sup-

pressed in denoting the evaluation homomorphism; one simply writes eval(a, T) or even briefer $\mathrm{eval}_a(T)$. In the special case when T is a polynomial, the evaluation algebra A usually coincides with the coefficient domain.

The concept of *polynomial function* is defined in terms of the evaluation homomorphism, so it stays the same as before: For a polynomial $p \in \mathcal{P}_{\mathfrak{V}}(A, \{x_1, \ldots, x_n\})$ over the coefficient domain $A : \mathfrak{V}$, its associated polynomial function $\tilde{p} : A^n \to A$ is defined by $\tilde{p}(a) = \mathrm{eval}_{\mathfrak{V}}(a, p)$. The collection of all such polynomial functions forms an algebra in $\mathfrak{V}$, which we denote by $\mathcal{P}^n_{\mathfrak{V}}(A)$. For the commutative polynomials, this is just $\mathcal{P}^n(A)$ according to our previous notation (see Section 2); so we simply suppress $\mathfrak{V}$ when it defaults to the variety *UniCommRing*. Just like $\mathcal{P}_{\mathfrak{V}}(\_, X)$, the construction $\mathcal{P}^n_{\mathfrak{V}}(\_)$ is again a functor constructing the domain of polynomial functions from a given evaluation domain in $\mathfrak{V}$. Although we will only need it in the polynomial setting, we should also mention that everything goes through without any change, if one considers an arbitrary free algebra $\mathcal{F}_{\mathfrak{V}}$ is a variety $\mathfrak{V}$. However, the resulting "polynomial functions" in this case are usually called "derived operations".

We are now finished with the construction of polynomials according to the old definition. This has taken a considerable portion of time (and paper), but we think that it pays off: We have placed the 'real idea' of polynomials—in a sense that will become clear soon—into its proper mathematical setting, thereby uncovering some deeper principles of how structures are built up in mathematics. In the next subsection, we will address some algorithmic questions: not surprisingly, this will lead us back to the succinct definition.

A.4 Computing with Polynomials

Up to now, we have not lost any thought about the *algorithmic aspects* of the old definition of polynomials. Let us analyze this in the well-known situation of commutative polynomials. A polynomial like $x^2 + 2$ is now a congruence class

$$\{\mathrm{x}^2 + 2 - 1,\ \mathrm{x}^3 + \mathrm{x}^2 - \mathrm{x}^3 + 2,\ (3 - 2)\,\mathrm{x}^2 + (-(-2)),\ \ldots\} \tag{73}$$

containing infinitely many terms all representing the same polynomial $x^2 + 2$. This may be a nice characterization for theoretical investigations, but it is obviously quite useless for practical computations. The succinct definition, where the polynomial is conceived as the finite sequence $\langle 1, 0, 2\rangle$, is very pragmatic compared with the infinite set (73). All the necessary operations can be carried out with this finitary data structure, and they are actually rather simple to implement. So it seems the old definition has no chance of surviving in our modern times of emphasized computer implementation? Let us reconsider the situation for a moment.

The formalization of polynomial domains as free algebras was actually *following some old intuitions*. Now the old mathematicians have done computations with polynomials for a long time, without 'having' any precise definition—neither the old nor the succinct one. So what did they do? Let us once again look at the computation (52), namely

$$(2 + \mathrm{x})\,(-2 + \mathrm{x}) = -4 + 2\,\mathrm{x} - 2\,\mathrm{x} + \mathrm{x}^2 = -4 + \mathrm{x}^2, \tag{74}$$

from a naive and 'innocent' viewpoint: We multiply the polynomials $2 + x$ and $-2 + x$ by considering them as "formal expressions in an indeterminate x", thus obtaining the formal expression $(2 + x)\,(-2 + x)$, which we "rewrite" first into the formal expression $-4 + 2\,x - 2\,x + x^2$ and then into $-4 + x^2$. The latter "is" then the resulting polynomial. What has happened here? Obviously, the rewrite steps carried the formal expressions along the congruence $\mathrm{Eqv}_{\Sigma,\Phi}(X)$ induced by the

axioms Φ in the signature Σ of the variety *UniCommRing*. So the "formal expressions" are nothing else than terms representing the corresponding congruence class. We will call such representing terms *polynomial terms*, whatever the variety $\mathrm{Cat}_{\Sigma,\Phi}$ may be. Note, however, that they are called "polynomials" on page 84 of the Universal Algebra text [20]!

So the basic idea is that one can perform effective computations on the polynomial forms rather than the polynomials themselves. But how do we decide whether two polynomials are equal then? Look at the polynomial (73) again: Some computation might yield the polynomial form $x^2 + 2 - 1$ as its result, another computation $x^2 + 2$. How can we know that they represent the same polynomial? In other words, the general problem is to find a decision algorithm for the congruence relation $\mathrm{Eqv}_{\Sigma,\Phi}(X)$ in order to decide equality in $\mathcal{F}_{\Sigma,\Phi}(X)$. In the literature about rewriting, it is called the *word problem*, and there several are well-known instances where it is unsolvable; see page 59 in [1]. Hence we cannot hope to control *every* kind of polynomial in this way, but it turns out that there are decision algorithms for many practically interesting polynomial types—of course including the commutative polynomials.

The fundamental idea is to fix the direction in which each axiom of Φ is used in the following way: It should not matter in which order any one of several applicable axioms is used for rewriting a given term; in the end, one should always come up with the same final term. Rewrite systems of this type are called *convergent*, and the two key requirements listed above are called confluence (the order does not matter) and termination (there is always a final term). See page 9 in [1] for precise definitions. Basically, one has to do the following: For a fixed axiom system Φ in a signature Σ and a set of indeterminates X, one defines a binary *reduction relation* $\longrightarrow$ on $\mathcal{T}_\Sigma(X)$ which expresses that some equation with a fixed orientation is used once on a term. Its reflexive, symmetric and transitive closure, denoted by $\overset{*}{\longleftrightarrow}$, coincides then with $\mathrm{Eqv}_{\Sigma,\Phi}(X)$. But what we need for computational purposes is of course the oriented version: the reflexive and transitive closure, called the corresponding *rewrite relation* and denoted by $\overset{*}{\longrightarrow}$. The art of rewriting is now to add some—logically speaking—redundant equality axioms to Φ such that the corresponding rewrite relation is indeed convergent.

In fact, there is a well-known algorithm for realizing this so-called *completion* process for many practically interesting purposes due to Donald E. Knuth and Peter B. Bendix [39], called the Knuth-Bendix algorithm (although it may fail on some inputs and run forever on others, so one should rather call it a procedure). The main idea is that for obtaining a complete system, it suffices to add oriented equalities—called rewrite rules—for very specific situations, namely when two earlier rewrite rules overlap on their left-hand sides; the corresponding right-hand sides are then said to form a *critical pair*. The fundamental idea of critical pairs lies at the heart of many important algorithms in computer science, collectively known as critical-pair/completion or briefly CPC procedures [12]. Besides the Knuth-Bendix algorithm for solving particular word problems, there are two other famous instances of this idea, both discoveries being independent from Knuth and Bendix's: One is Buchberger's algorithm for computing Gröbner bases [17], the other (though slightly less in the CPC spirit) is Robinson's resolution procedure [55].

For our present purposes, we need not resort to sophisticated machinery; let us simply remark that by inspection, one can find convergent rewrite systems for many practically important types of polynomials—see [42] for the case of unital commutative rings, groups, lattices, and boolean algebras, including proofs of their convergence. For example, the rewrite system for the 'classical' *commutative polynomials* proceeds by expanding the polynomials (this corresponds to using the distributivity axiom directed 'forward' if both operands are non-constant), collecting equal monomials (this corresponds to using the distributivity axiom directed 'backward' if either operand is constant), simplifying the coefficients (using the operation-table axioms in the 'computation direction'), ordering the monomials and their indeterminates appropriately (using the

associativity/commutativity axioms of addition and multiplication in a direction prescribed by a suitable term ordering), and so on. A similar strategy can be applied for unital rings (without commutativity of multiplication!), whose variety we shall denote by *UniRing*; the only difference is that we need not order the indeterminates within monomials. From now on, we will simply speak of *uncommutative polynomials* (polynomial terms, polynomial forms) in this case. This name is a bit ugly, but we will not have to use it often because we will mainly work with the slightly different notion of *non*commutative polynomials to be introduced in Section 5.

Whenever one has a convergent rewrite system $\overset{*}{\longrightarrow}$ for a congruence $\sim$ (meaning $\overset{*}{\longleftrightarrow}$ coincides with $\sim$), one can rewrite a given term until this process stops (since convergence guarantees termination); the resulting term is then called the *normal form* of the input term, and it is unique due to the confluence property (which is also guaranteed by convergence). Denoting the normal form of a term t by $t_\downarrow$, the corresponding *normalization* mapping $t \mapsto t_\downarrow$ has two essential properties: First, it preserves the congruence, meaning $t \sim t_\downarrow$, because $\overset{*}{\longrightarrow}$ is clearly a subrelation of $\overset{*}{\longleftrightarrow}$. Second, it 'equalizes' the congruence, meaning $t \sim u \Rightarrow t_\downarrow = u_\downarrow$, because of the uniqueness property. A mapping with these two properties is called a *canonical simplifier* for $\sim$, and the corresponding normal forms are then canonical forms for $\sim$; see page 12 in [16]: The canonicality requirement is equivalent to the decidability of the given congruence and hence to the solvability of the word problem in the free algebra $\mathcal{F}_{\Sigma,\Phi}(X)$. Furthermore, the factor domain $\mathcal{F}_{\Sigma,\Phi}(X)$ is then isomorphic to the corresponding *ample algebra*: Its universe is the collection of all the normal forms; its operations proceed by first applying the operations as in $\mathcal{T}_\Sigma(X)$ and then normalizing the resulting terms; see page 13 ibidem.

Applied to polynomial terms, this solves the problem of effective computation in all those cases, where we have a convergent rewrite system. Following [5] on page 58, the normal form of a given polynomial term will be called its associated *polynomial form*. Then we can interpret the above isomorphism as saying that we can basically identify the polynomials with their associated polynomial forms. For example, the polynomial (73) corresponds to the polynomial form $x^2 + 2$. Now we can also see the *connection to the* succinct *definition* of polynomials: Since we can always represent a polynomial form in $R[X]$ by the finite multisequence of its coefficients, the 'succinct polynomials' turn out to be nothing else than custom-tailored data structure of the corresponding polynomial forms, including an addition and multiplication that is optimized for this data structure. It is therefore clear that the succinct concept of polynomials is so well suited for the algorithmic tasks of today's computer algebra—limited, though, to the special case of polynomials for the variety *UniCommRing*, which is of course the most important type of polynomials occurring in practice. In the general case, it is highly non-trivial to come up with similar custom-tailored data structures and optimized operations on them. In the literature, this problem is known as *term indexing*.

For us, the most important case will not be the commutative but—in some sense—the *uncommutative polynomials*. So let us briefly consider the problem of indexing for this situation. Taking away the commutativity for multiplication in the axiom system (67), there are seven axioms for the variety *UniCommRing*; we will discuss their use one by one. Obviously, it is still possible to expand a polynomial into a sum monomials by virtue of the distributivity axiom. By the associativity and commutativity of addition, so we can then collect the monomials in a set, realized as a list with a fixed term ordering. The law of the additive neutral is realized by simply taking out empty monomials from this list. Translating $-x$ into $(-1)\,x$, the law of the additive inverse becomes superfluous as it is subsumed by the operation table. The associative law for multiplication tells us that we can regard the monomials as words, having indeterminates and coefficients as letters (note that the coefficients do *not* in general commute with the indeterminates—see the next subsection). The law of the multiplicative neutral just tells us to discard empty words. Finally, the operation

table may be applied whenever one meets a subterm without indeterminates. Hence we can conclude that the uncommutative polynomial forms are isomorphic to the data structure of *word lists*.

Unlike to the situation of commutative polynomials, however, we do not gain anything from using the isomorphic data structure. All we have to do is to regard addition and multiplication as *flexible operations*; see [41]. In this view, the unary versions act as identity functions, so $\mathrm{Plus}(A) = A$ and $\mathrm{Times}(A) = A$. Furthermore, the nullary versions are the corresponding neutral elements, so $\mathrm{Plus}() = 0$ and $\mathrm{Times}() = 1$. Obviously it makes no difference whether we understand a list $\langle A, B, C\rangle$ as a data structure like $\mathrm{List}(A, B, C)$ with the flexible constructor "List" or as an addition $\mathrm{Plus}(A, B, C)$ with the flexible constructor "Plus". But note that "Plus" has the additional role of *set union* in this case, meaning that it joins two lists and then reorders it according to the fixed ordering. For the multiplication, the situation is similar. It amounts to the same thing whether we consider a word $A\,B\,C$ as a data structure $\mathrm{Word}(A, B, C)$ with the flexible constructor "Word" or as a multiplication $\mathrm{Times}(A, B, C)$ with the flexible constructor "Times". The difference to the previous case of lists is that "Times" now acts as *concatenation* on the words, which does not involve any reordering of the letters—reflecting the uncommutative nature of this kind of polynomials.

Therefore we conclude that the advantages of the succinct definition are lost upon moving from commutative to uncommutative polynomials: In our opinion, it is more reasonable to carry out the computations immediately on the uncommutative polynomial forms rather than introducing some artificial definition similar to what is done for commutative polynomials in computer algebra—where it clearly does make sense.

A.5 Commutative versus Noncommutative

There is one more subtlety we must take into account. In the upcoming applications of uncommutative polynomials, we will encounter situations where the coefficients are taken from a commutative ring or even from a field. But for uncommutative polynomials as we defined them, there will be *no commutation between indeterminates and coefficients*, even though the coefficients may commute amongst themselves. In typical situations, we are dealing with the complex field $\mathbb{C}$ as a *commutative* coefficient domain, and the indeterminates $X = \{x_1, \ldots, x_n\}$ will represent differential, integral and multiplication operators. As a concrete example, let us look at the situation $X = \{\sin, \partial\}$, where $\sin$ represents the multiplication operator mapping a function f to the function $x \mapsto f(x)\sin x$ and ∂ the differentiation operator mapping a function f to its derivative f'. The corresponding domain of differential operators is then given by the *uncommutative* polynomial ring $\mathcal{P}_{UniRing}(\mathbb{C}, \{\sin, \partial\})$, since $\sin$ and ∂ obviously do not commute. This is a severe problem because we would like to do computations like

$$\begin{aligned}&(3\,\partial + 2\sin)\,(7\,\partial + 4) = \\ &\quad (3\,\partial)\,(7\,\partial) + (3\,\partial)\,4 + (2\sin)\,(7\,\partial) + (2\sin)\,4 = 21\,\partial^2 + 12\,\partial + 14\sin\partial + 8\sin,\end{aligned} \tag{75}$$

where we have, among others, used the reduction

$$\begin{aligned}&(3\,\partial)\,(7\,\partial) \to 3\,(\partial(7\,\partial)) \to \\ &\quad 3\,((\partial 7)\,\partial) \overset{!}{\to} 3\,((7\,\partial)\,\partial) \to 3\,(7\,(\partial\,\partial)) \overset{*}{\to} (3 * 7)\,(\partial\,\partial) \to 21\,(\partial\,\partial) = 21\,\partial^2\,.\end{aligned} \tag{76}$$

All the steps indicated by $\to$ are just invocations of associativity, the step indicated by $\overset{*}{\to}$ appeals to the operation table for multiplying complex numbers, and the last equality is merely a

notational convention. The critical step is the one indicated by $\overset{!}{\to}$, where we have made use of the commutation between ∂ and 7. Of course, we need these commutations between all indeterminates and all coefficients. Speaking a bit more generally, we are asking for a *new type of* '*uncommutative polynomials*' over a commutative ring R in the indeterminates X, where $x\,r = r\,x$ for all indeterminates $x \in X$ and all coefficients $r \in R$. Maybe one only has to adjust the axioms of the variety *UniRing* slightly for constructing this new type of 'uncommutative polynomials'?

A moment's thought, however, reveals that such 'uncommutative polynomials' *cannot exist* in the (very broad!) understanding of the polynomial concept outlined in the preceding subsections. The reason is simply that the indeterminates would not be 'indeterminate' anymore; they would not be fresh constants since the commutations bind special properties to them. For a clear concept of polynomial, we should keep this requirement such that the term "indeterminate" deserves its name: Their essential idea is that we know nothing about them—except that they are distinct anonymous elements of the carrier.

Let us make this point clearer by way of another example. As a coefficient domain, we take $\mathbb{C}$ as before. But the indeterminates should be $X = \{\sin, \mathfrak{c}\}$ now, where sin is as before and $\mathfrak{c}$ is the 'cubing operator' mapping a function f to the cubed function $x \mapsto f(x)^3$. Again we can consider the uncommutative polynomial ring $\mathcal{P}_{UniRing}(\mathbb{C}, \{\sin, \mathfrak{c}\})$, but now it is clear that we would not want to have a commutation between the indeterminate $\mathfrak{c}$ and, say, the coefficient 3: The operator $\mathfrak{c}\,3$ maps f to $x \mapsto (3\,f(x))^3 = 27\,f(x)^3$, whereas the operator $3\,\mathfrak{c}$ maps f to $x \mapsto 3\,f(x)^3$. Since the *commutations do not work* for this particular interpretation of the indeterminates, we cannot expect that they should work in general; the polynomial ring $\mathcal{P}_{UniRing}(\mathbb{C}, X)$ cannot 'know' anything about the particular interpretation we have in mind!

What we can learn from this example is that one typical interpretation of the rings $\mathcal{P}_{UniRing}(\mathbb{C}, X)$ is various *composition rings* (meaning that composition plays the role of the ring multiplication; see page 306 in [53]) of operators acting on certain complex functions. The special situation with commutations between indeterminates and coefficients arises when we restrict our attention to *linear operators*: In this case, we have $x(f + g) = x(f) + x(g)$ and $x(r\,f) = r\,x(f)$ for all indeterminates $x \in X$, for all coefficients $r \in R$, and for all admissible functions f, g we have in mind. The only interesting information we can extract from the first property to the polynomial ring is by substituting for f and g functions $y(\hat{f})$ and $z(\hat{g})$ arising from other operators $y, z \in X$ and arbitrary admissible functions $\hat{f}$ and $\hat{g}$. This gives $x\big(y(\hat{f}) + z(\hat{g})\big) = x\big(y(\hat{f})\big) + x(z(\hat{g}))$ for arbitrary admissible functions $\hat{f}$ and $\hat{g}$. Since operator composition is just polynomial multiplication in this interpretation, this amounts to $x(y + z) = x\,y + x\,z$, which is always fulfilled due to the distributive law of the variety *UniRing*. The second property means that we require $x\,r = r\,x$ for all indeterminates $x \in X$ and for all coefficients $r \in R$.

Since we cannot construct a new type of 'uncommutative polynomials' for modelling such domains of linear operators as are relevant to us, let us consider a different device for enforcing the desired commutations. In fact, we might have various other desires coming from specific interpretations of the indeterminates, e. g. the property $\partial^2 \sin = -\sin$ in the example presented at the beginning of this section. Now equalities like $\partial^2 \sin + \sin = 0$ or $\partial 3 - 3\,\partial = 0$ are all of the form $p = 0$, where p is some uncommutative polynomial. It is a well-known fact from ring theory that one can always form the *quotient ring* of all the polynomials with respect to the *ideal* generated from one or more such left-hand side polynomials p; this is true for the commutative and uncommutative case alike.

It is advantageous to distinguish between ideals associated with fundamental properties like the commutation ideal $\mathfrak{Q}(R, X) = (r\,x - x\,r \mid x \in X \wedge r \in R)$ for an arbitrary uncommutative polynomial ring $\mathcal{P}_{UniRing}(R, X)$ and other more 'accidental' ideals like the above example $\mathfrak{I} = (\partial^2 \sin + \sin)$. In

the first case, it seems appropriate to introduce a new name for the resulting domain $\mathcal{P}_{UniRing}(R, X)/\mathfrak{Q}(R, X)$, which does now indeed provide the desired 'uncommutative polynomials with indeterminate-coefficient commutations'. Following the literature, we will call it the ring of *noncommutative polynomials* or briefly the noncommutative polynomial ring and denote it by $R\langle X\rangle$; see page 527 in [3]. Analogous to the commutative case, we will abbreviate $R\langle\{x_1, \ldots, x_n\}\rangle$ by $R\langle x_1, \ldots, x_n\rangle$, as it is also customary in practice.

Having constructed this new domain, we should again pose the question of *term indexing*. Obviously, a noncommutative polynomial like $21\,\partial^2 + 12\,\partial + 14\sin\partial + 8\sin$ occurring as the result in (75) can be described as a set (or ordered list—see Section 4) containing pairs with a coefficient and a word whose letters are only indeterminates. In the example, this list would be

$$\{\langle 21, \partial\partial\rangle, \langle 12, \partial\rangle, \langle 14, \sin\partial\rangle, \langle 8, \sin\rangle\}. \tag{77}$$

Now we can always view a finite set of pairs as a function with finite domain, if the correspondence between both sides of the pairs is unique in some direction. In examples like (77), there will always be only one coefficient for a given word—simply because the normal form presupposes that all like monomials are fused. Hence we can view the noncommutative polynomial forms as finite functions from the word monoid X^* to the coefficient ring R. In fact, we can always think of such a function as defined on the whole of X^* by mapping all the missing words to the coefficient 0, and so we will identify the noncommutative polynomial forms with the collection of all functions from X^* to R having finite support (but with the coefficients written in the left component). In algebra, the construction of all finitely-supported functions from a given monoid M to a ring R is known as the *monoid ring* over R and M; it is an important tool in representation theory, where M is usually even equipped with a group structure and is accordingly called the group ring; see page 246 in [53]. So the support of a polynomial $p \in R\langle X\rangle$ is $\{w \in X^* \mid p(w) \neq 0\}$, and we will denote it by $\mathrm{supp}(p)$.

Since it is well-known that the word monoid X^* is coincides with the free monoid over X, one often reads the formulation that $R\langle X\rangle$ is the monoid ring over R and the free monoid over X; see page 527 in [3]. Analogously, one can characterized $R[X]$ as the monoid ring over R and the free abelian monoid over X; see page 64 in [3]. The latter is nothing else than the monoid of finite multisequences, so the polynomials of $R[x_1, \ldots, x_n]$ are just finitely-supported functions from $\mathbb{N}^n$ to R as described in Section 2. However, one should keep in mind that only $R[X]$ but not $R\langle X\rangle$ is a 'real' polynomial domain in the authoritative sense of the old definition. Having issued this warning, we will from now on follow the common practice of using the name "polynomial" also for the elements of some quotient $\mathcal{P}_{\Sigma,\Phi}(A, X)/\mathfrak{Q}$ of a given polynomial domain $\mathcal{P}_{\Sigma,\Phi}(A, \Phi)$. Here Φ is an equational axiom system in a signature Σ, A is an algebra from $\mathrm{Cat}_{\Sigma,\Phi}$, X is a set of indeterminates, and $\mathfrak{Q}$ is a congruence on (for rings: determined by an ideal in) $\mathcal{P}_{\Sigma,\Phi}(A, X)$.

We should also mention that one can consider $R[X]$ or $R\langle X\rangle$ as an algebra over R, which is then called the *monoid algebra* over the free monoid over X or the free abelian monoid over X, respectively. Note that the term "algebra" is here used in the narrower sense of what is also known as "bilinear algebra" [49], namely a ring which is at the same time a compatible module (compatibility means mixed associativity of multiplication). Considering this algebra structure, one may also characterize $R[X]$ or respectively $R\langle X\rangle$ as the free abelian algebra over X or as the free algebra over X; see pages 448ff in [6]. This fact also justifies the parallelism suggested by the notations $R[X]$ and $R\langle X\rangle$.

As we have now seen that the monoid ring over R and X^* provides the appropriate data structure for noncommutative polynomials over R in X, let us now ask ourselves again how this should be handled in the *implementation*. Obviously, the only change compared to the uncommutative polynomials is that the words contain only indeterminates, whereas the coefficient is separated as

an additional component of the pair comprising a monomial. Hence we still need not change anything, as long as we agree that the coefficient should always be the first element—even if it is 1—in the word expressed by Times(…). For example, the noncommutative polynomial in (77) would be represented as

$$\text{Plus}\,(\text{Times}\,(21,\,\partial,\,\partial),\,\text{Times}\,(12,\,\partial),\,\text{Times}\,(14,\,\sin,\,\partial),\,\text{Times}\,(8,\,\sin)), \tag{78}$$

which is nothing else than the corresponding *noncommutative polynomial form* $21\,\partial^2+12\,\partial+14\sin\partial+8\sin$, the only difference being in the concrete syntax used for writing it. So also in this case, there is no disadvantage if we carry out the computations in the style of the old definition, as long as we treat the first letter of each word in a special way that reflects its role as the coefficient of the monomial.

Epilog: Prospects of Generalization

Judging from the applied point of view, what we have presented in this PhD thesis is of course not—yet—very exciting. We have only considered a rather *narrow and simple class of BVPs*, namely regular ones for differential equations that are simultaneously ordinary, linear, even having constant coefficients, and having only one unknown. However, we believe that there are some prospects for generalizing our approach. Naturally, the work necessary for this will become increasingly more difficult as one climbs up the ladder of generalizations; but we hope this work will be rewarded by a proportional increase of deep mathematical substance.

Let us first look at some straight-forward lines of *generalization* (we have discussed most of these also in [57]):

- The first restriction that one should try to get rid of is that of constant coefficients, thus widening the scope to *general linear differential operators*. It should be possible to find out how this can be done by comparing with the well-established function-level method of described e.g. on page 189 in [38]. The only step to be adapted here is the right inversion of the given differential operator, which obviously does not any more factor into independent linear factors. But knowing the fundamental system, it may still be possible to successively split off and invert "dependent linear factors" from the right until the given differential operator is exhausted.

- We can view *systems of differential equations* (together with their boundary conditions) instead of a single one. In the linear case, the resulting theory is very similar to scalar BVPs, using a Green's matrix instead of a Green's function; see e.g. page 249 in [38]. Our method should be extensible to this case in a fairly simple manner. In the worst case, we have to recede to our original approach [57] via Gröbner bases and adapt them to work for vectors of polynomials rather than single ones. Essentially this amounts to computing Gröbner bases in modules, which is a routine task for commutative polynomials (see e.g. pages 485ff in [3]) and should smoothly carry over to noncommutative ones.

- It is certainly a much greater challenge to move from ordinary to *partial differential equations*. In principle, the algebraization employed in our approach extends in a straightforward way, e.g. introducing D_x and D_y instead of the single differentiation D and analogous operators for integration. Here one might be able to benefit a lot from the algebraic approach employed in Riquier–Janet theory and from the symmetry methods of Lie analysis. The treatment of boundary values must of course be adapted. Besides this, the analog of right inversion will be far more complex for most partial differential operators; it might be analogous to the elimination techniques used in the holonomic approach [71].

- One of the most difficult generalizations is probably the step towards *nonlinear* BVPs. The reason is that our algebraic model does not lend itself easily to describe nonlinear differential operators, and a systematic approach might lead to general rewriting (still with respect to the polynomial congruence), where one needs substitution in addition to replacement. Maybe this could be handled by a suitable combination of Gröbner bases and the Knuth-Bendix algorithm; see [2] and [48].

- In this thesis we have only considered regular BVPs in the sense that there is a unique solution, and in this case the Moore–Penrose inverse coincides with the actual inverse. If

the BVP is *underdetermined*, however, one can still search for a so−called modified Green's function; see page 215f in [63]. Since the modified Green's function just corresponds to a Moore−Penrose inverse, our method should be adaptable to this case in a natural way.

- As a kind of curiosity, we should also be able to handle certain *integro−differentialequations*. In fact, the Green's algebra provides a uniform way of expressing integral as well as differential equations—and their mixtures.

Beyond these rather direct continuations of the research topic treated in this thesis, we believe that our approach has some intrinsic interest not directly tied to BVPs of any kind. The essence of our method can be described as solving problems at the operator level via polynomial methods. This could be a new research paradigm applicable to various problems of a field that might be called *symbolic functional analysis*. Up to now, symbolic methods have conquered the following two "main floors": numbers (computer algebra) and functions (computer analysis); naturally, the third floor would be: operators (symbolic functional analysis). We have described these ideas in more detail in [9]; so let us just mention here two examples of problems residing on this third floor:

- Certain problems in *potential theory* seem to have a flavor that is very similar to that of BVPs for PDEs, at least when seen from the symbolic viewpoint. It is therefore natural to ask in how far one could transfer some ideas from BVPs to the potential setting. In particular, one would like to formulate an algebraic setup that allows to express the operator induced by the potential function (analogous to the Green's operator induced by the Green's function).

- The field of *inverse problems* opens a whole arena of possible applications for methods of symbolic functional analysis. Even though one cannot usually expect algebraic solutions for such problems, the polynomial approach will certainly uncover a great deal about the solution manifold. In particular, it might be possible to transform the given problem into a different one possessing more profitable properties.

As the time available for producing and writing a PhD thesis is always terribly short, we do hope that it will be possible to develop some of these interesting ideas in the future. As a matter of fact, (good) mathematics is never finished; it always opens new doors.

Curriculum Vitae

Affiliation

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, A-4040 Linz, Austria
Tel. ++43 732 2468 9926
Fax ++43 732 2468 9930
Markus.Rosenkranz@risc.uni-linz.ac.at

Personal Information

Name:	Markus Rosenkranz
Date and Place of Birth:	September 5, 1971 in Wels / Upper Austria
Nationality:	Austrian

Education

1982-1990	Humanistisches Stiftsgymnasium Kremsmünster
Sep 87-Jul 88	High school year in Sacramento / California
Jun 90	Matura (with excellence)
1991-97	Technische Mathematik, Lehramt Mathematik / Physik in Linz
May 97	Final exam Technische Mathematik (with excellence)
Jan 98	Final exam Lehramt Mathematik / Physik
Oct 97-May 98	Military Service

Career History

Jun 98-Apr 00	Engagements in the field of industrial mathematics
Jun 98-Oct 98	Institute for Industrial Mathematics (Christian-Doppler Lab)
Nov 98-Oct 99	Company MathConsult
Nov 99-Apr 00	Competence Center for Industrial Mathematics
Apr 00-May 00	Translation project for Springer (dynamic geometry)
May 00-Sep 03	PhD student at RISC in the *Theorema* group

Career Related Activities

Jun 98	Fluid mechanics seminar with Fluent™ in Darmstadt
Oct 98	Fluid mechanics seminar with Fire™ in Graz
Apr 99	Seminar on finite-element implementations with Diffpack™
in Oslo	
Sep 99	UniSoftware-Plus seminar for C++
Apr 02	Visiting Young Researcher at Nijmegen University

Academic Degrees

Diploma Degree for Technical Mathematics, Johannes Kepler University of Linz, May 1997.
Title of the Diploma Thesis: *Lagrange Inversion*.

Master Degree for Pedagogical Mathematics, Johannes Kepler University of Linz, January 1998.
Title of the Diploma Thesis: *The Joy of Combinatorics.*

Publications

The journal article [57].

The extended abstract [60], which was presented as a poster at ISSAC 2003.

The talks [59], [58], [56].

Acknowledgements

The work described in this thesis has been carried out in the frame of the *Theorema* project at the RISC institute, which is supported most prominently by the "Spezialforschungsbereich for Numerical and Symbolic Scientific Computing" (SFB F013) at the University of Linz, the "PROVE Project" supported by the regional government of Upper Austria, and the european union "CALCULEMUS Project" (HPRN-CT-2000-00102).

As described in the Prolog, this PhD thesis is the fruit of an interesting cooperation between a symbolic group of the University of Linz—led by B. Buchberger—and a numeric group of the same university—led by Heinz W. Engl—started around October 2001, and I am grateful that both of these professors are also *supervisors of the present thesis*. Though very natural from the contents point of view, this is not at all easy for them to accommodate in their tight schedules! So I am all the more grateful to them.

It would only be half of the truth if I mentioned only the issue of supervision in view of B. Buchberger and Heinz W. Engl. I have also *learned a great deal* from them about how to do math and how to manage science. Heinz W. Engl (whom I know especially from my engagements in industrial mathematics—see the above curriculum vitae) impressed me by his strong will and consequent strategy for settling all kinds of enterprises; besides this, I have always enjoyed his lectures combining deep substance and formal rigor. Being a PhD student of B. Buchberger for three years, I had the special pleasure of absorbing a lot of deep "mathematical wisdom" that has significantly changed my own outlook at mathematics. Moreover, I continue to be amazed by his unbelievable versatility: discussing deep mathematical theorems, playing a clown for his five-year old daughter, conducting a meeting with the software park managers—all this within a few hours, sometimes almost in parallel…

In fact, I could say similar things about many people that I have come to know at RISC during the last three years—staff and postdoc and PhD students alike. The *atmosphere at RISC* is somehow unique (I hope this will not be lost in the next years!), maybe because a considerable number of RISC people live in Hagenberg and share some of their spare time; maybe because the village air contributes to a very special research setting (though one must admit that the "village" is expanding with awesome speed); maybe it is simply the people working here—who knows? Anyway, I would really like to thank them all!

Within the larger group of RISC, it was obviously the *Theorema* group, which was my nearer academic environment in the past three years. And I have definitely enjoyed being part of them. Let me try to give a short characterization of this group as I see it (names are obviously ordered alphabetically):

- *Adrian Craciun*—the big guy in black, always a smile (or a grin?) on his face. Never short of some—all too often cynical—jokes!
- *Tudor Jebelean*—our great and strong pillar, when Bruno is not around (which does happen sometimes!), taking nearly any calamity from the humorous side, with practical advice immediately to follow. Sometimes I thought he is predicate logic alive.
- *Gabor Kusper*—now in Hungary, for the purpose of marrying and leading a software group and such things; we hope to see him again! His mild nature is liked by everyone.

- *Koji Nakagawa*—he is our logico-graphic symbol and a Japanese *par excellence*: Taking pictures even of the food at mensa, of course using an ultra-tiny Japanese digital camera! His kind advice for all kinds of software technicalities is always appreciated—e.g. in the bibliography of this thesis.

- *Florina Piroi*—the proof object of the group. In fact, she has always proved to be a reliable friend and a strong core developer of the internal engine used in *Theorema*. We admire her joyful spirit and the many practical skills she has in life.

- *Nikolaj Popov*—my nearest desk-neighbor and my closest friend here. I cannot count how often he has helped me in all kinds of problems, and many other *Theorema* members can say the same of him! And I always enjoy our discussions about our research ideas in mathematics and computer science.

- *Judit Robu*—our periodic geometry expert, though we have not seen her for a while now. When she is here, it is almost canonical and always a pleasure to have lunch together. Hopefully we will see her again!

- *Temur Kutsia*—the man with the equality sign in his hands. No, he is really very knowledge able in equational reasoning, and I am grateful for his help with the proof of Theorem 17 of Chapter 1—although in the end it did not work this way. And we all appreciate his special gentle-dry (typically Georgian?) humor.

- *Christian Vogt*—who has left us all too soon, we really wish you a good time in Innsbruck! Yes, and I do remember all those exciting discussions we had on the vision of axiomatization and formalization. I am looking forward to see him again and somehow resume our discussions.

- *Wolfgang Windsteiger*—so to say *Mathematica* in person, at least in our group! Lately, we could also call him Mr. Set Theory… Being the "software director" of our group, his efforts for a consistent and elegant system are really enormously valuable, and he is never hesitant to provide help in all issues of *Theorema* and *Mathematica*—which was his greatest problem when he was still sitting in my room!

In addition to the broad general support I enjoyed from all these people, there were also some *specific pieces of advice* from which I benefitted considerably. I will try to remember them here but if I have forgotten any, please forgive me and just add your name silently at this place. So let me express my special thanks to these people:

- *James B. Cooper* pointed me to Silva's definition of distributions and showed me how to characterize boundary values in this frame; see the explanation after Definition 37 in Chapter 1.

- *Ralf Hemmecke* helped me with some issues related to Theorems 17 and 28 of Chapter 1; see also the proof of the former theorem.

- *Erik Hillgarter* explained me several things about how differential operators are handled in Lie analysis.

- *Aleksey Kondratyev* helped me with the proof of Theorem 33 of Chapter 1.

- *Arnold Neumaier* showed me how to characterize boundary values in the classical setting of distributions; see again the explanation after Definition 37 in Chapter 1.

Last but not least, I am of course obliged to my *family* at home: my mother, sister, grandmother, *in-spe* brother-in-law, and—even though he is not with us any more—my father. It is all too easily overseen what a gift it is to call some place one's home, and I am indeed very grateful for this.

Finally and most fundamentally, let me thank *God* as my wonderful Father, who has given me my place on Earth and many good gifts of all kinds.

References

[1] Franz Baader, Tobias Nipkow. *Term Rewriting and All That*, Cambridge University Press, Cambridge (UK), 1999 edition.

[2] L. Bachmair, H. Ganzinger. Buchberger's algorithm: A constraint–based completion procedure. In J.-P. Jouannaud (Editor), *First International Conference on Constraints in Computational Logics*, volume 845 of Lecture Notes in Computer Science, page 285-301. Springer, New York, 1994.

[3] Thomas Becker, Volker Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*, series Graduate Texts in Mathematics. Springer, New York, 1993.

[4] George M. Bergman. The Diamond Lemma for Ring Theory. *Advances in Mathematics*, 29(2):179-218, August 1978. Hardcopy number {19}.

[5] Garret Birkhoff, Saunders MacLane. *A Survey of Modern Algebra*, The Macmillan Company, New York, 1958. Revised edition.

[6] Nicolas Bourbaki. *Elements of Mathematics Construction: Algebra I*, Springer, New York, 1989.

[7] Bruno Buchberger. An Algorithmic Criterion for the Solvability of Algebraic Systems of Equations (German). *Aequationes Mathematicae*, 4:374-383, 1970.

[8] Bruno Buchberger. *Categories and Functor Session*. Personal communication during a private seminar, December 2001.

[9] Bruno Buchberger, Heinz W. Engl. *Computer Algebra for Pure and Functional Analysis*. An FWF Proposal for a New SFB Project (F1322?), 2003.

[10] Bruno Buchberger. *Functor Programming in Mathematica*. Colloquium Talk (University of Delaware, Department for Information and Systems Sciences, USA), May 24, 1996.

[11] Bruno Buchberger. *Functors for Mathematics*. Talk at the Conference for Computer Algebra and Algebraic Geometry (Dagstuhl, Germany), May 26, 1997.

[12] Bruno Buchberger. History and Basic Features of the Critical–Pair/Completion Procedure. In Jean-Pierre Jouannaud (Editor), *Rewriting Techniques and Applications*, Reprinted from the Journal of Symbolic Computation (Volume 3, Numbers 1 & 2, 1987) page 3-38. Academic Press, London, 1987.

[13] Bruno Buchberger. Introduction to Gröbner Bases. In Bruno Buchberger, Franz Winkler (Editors), *Gröbner Bases and Applications*, volume 251 of London Mathematical Society Lecture Note Series, page 3-31. Cambridge University Press, Cambridge (UK), 1998.

[14] Bruno Buchberger, Franz Lichtenberger. *Mathematik für Informatiker*, Springer, Berlin, 1981.

[15] Bruno Buchberger. *Logic, Mathematics, Computer Science: The Accumulated Thinking Technology of Mankind*. Talk given at LCMS'2002.

[16] Bruno Buchberger, Rüdiger Loos. Algebraic Simplification. In Bruno Buchberger, G. E. Collins, Rüdiger Loos (Editors), *Computer Algebra: Symbolic and Algebraic Computation*, Springer, Wien, 1982.

[17] Bruno Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of Zero-Dimensional Polynomial Ideal (in German)*. PhD Thesis, Universität Innsbruck, Austria, 1965.

[18] Bruno Buchberger. *Theory Exploration with Theorema*. Second international Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNACS'00), Timisoara, Romania, 2000.

[19] Bruno Buchberger, Wolfgang Windsteiger. The Theorema Language: Implementing Object- and Meta-Level Usage of Symbols. In *Proceedings of Calculemus 98 (Eindhoven, Netherlands)*, 1998. Available as RISC–Report98-10, Research Institute for Symbolic Computation, Linz, Austria.

[20] Burris, Sankappanavar. *A Course in Universal Algebra*, Springer, New York, 1981. ISBN 0-387-90578-2.

[21] E. A. Coddington, N. Levinson. *Theory of Ordinary Differential Equations*, McGraw-Hill Book Company, New York, 1955.

[22] James B. Cooper. Personal communication, June 2003.

[23] Roy L. Crole. *Categories for Types*, series Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge (UK), 1993. ISBN 0-521-45092-6.

[24] H.-D. Ebbinghaus, J. Flum, W. Thomas. *Einführung in die mathematische Logik*, BI-Wissenschaftsverlag, Mannheim, 3rd edition, 1992.

[25] Samuel Eilenberg, Norman Earl Steenrod. *Foundations of Algebraic Topology*, Princeton University Press, Princeton, New Jersey (US), 1966.

[26] Heinz W. Engl, M. Hanke, Andreas Neubauer. *Regularization of Inverse Problems*, Kluwer, Dordrecht, 1996.

[27] Heinz W. Engl, M. Zuhair Nashed. New Extremal Characterizations of Generalized Inverses of Linear Operators. *Journal of Mathematical Analysis and Applications*, 82:566-586, 1981.

[28] Melvin Fitting. *First-Order Logic and Autmated Theorem Proving*, Springer, New York, 1990.

[29] Joachim von zur Gathen, Jürgen Gerhard. *Modern Computer Algebra*, Cambridge University Press, Cambridge (UK), 1999. ISBN 0-521-64176-4.

[30] Martin Giese. Integriertes automatisches und interaktives Beweisen: die Kalkülebene. Master Thesis, Institut für Logik, Komplexität und Deduktionssysteme, University of Karlsruhe, Germany, June 1998. Archived under URL[file:///home/marcus/private_html/references/logic/-Giese.IntegriertesAutomatischesUndInteraktivesBeweisen.ps].

[31] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:305-314, 1930. In: Karel Berka, Lothar Kreiser. Logik-Texte: Kommentierte Auswahl zur Geschichte der modernen Logik. Akademie-Verlag, Berlin, third edition, 1983.

[32] R.L. Graham, D.E. Knuth, O. Patashnik. *Concrete Mathematics*, Addison-Wesley, Reading, Massachussetts (USA), 1989.

[33] G. Grosche, V. Ziegler, D. Ziegler. *Teubner-Taschenbuch der Mathematik 1*, Teubner, Stuttgart, 1996.

[34] J. William Helton, Mark Stankus, John Wavrik. Computer Simplification of Engineering Systems Formulas. *IEEE Trans. Autom. Control*, 43(3):302-314, 1998.

[35] J. William Helton, John Wavrik. Rules for Computer Simplification of the Formulas in Operator Model Theory and Linear Systems. *Operator Theory: Advances and Applications*, 73:325-354, 1994.

[36] David Hilbert, Paul Bernays. *Grundlagen der Mathematik (Volumes 1 & 2)*, Springer, Berlin, 1968.

[37] Klaus Jänich. *Analysis fuer Physiker und Ingenieure*, Springer, Berlin, 1983.

[38] Erich Kamke. *Differentialgleichungen: Lösungsmethoden und Lösungen (Volume 1)*, Teubner, Stuttgart, tenth edition, 1983.

[39] Donald E. Knuth, Peter B. Bendix. Simple Word Problems in Universal Algebra. In J. Leech (Editor), *Proceedings of the Conference on Computational Problems in Abstract Algebra (Oxford 1967)*, page 263-298. 1970. Proceedings by Pergamon Press.

[40] Allan M. Krall. *Applied Analysis*, D. Reidel Publishing Company, Dordrecht, 1986.

[41] Teimuraz Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD Thesis (available as technical report 02–09) Research Institute for Symbolic Computation, Johannes Kepler University Linz, Castle of Hagenberg, Austria, 2002.

[42] Hans Lausch, Wilfried Nöbauer. *Algebra of Polynomials*, North-Holland, Amsterdam, 1973.

[43] J. L. Lions, E. Magenes. *Non-homogeneous Boundary Value Problems and Applications: Volume I*, Springer, Berlin, 1972.

[44] Rüdiger Loos. Introduction. In Bruno Buchberger, George E. Collins, Rüdiger Loos (Editors), *Computer Algebra: Symbolic and Algebraic Computation*, page 1-10. Springer, Wien, 1982 edition.

[45] Warren S. Loud. Some Examples of Generalized Green's Functions and Generalized Green's Matrices. *SIAM Review*, 12(2):194-210, April 1970.

[46] Saunders MacLane, Garrett Birkhoff. *Algebra*, The Macmillan Company, New York, 1967.

[47] Saunders MacLane. *Categories for the Working Mathematician*, volume 5, series Graduate Texts in Mathematics. Springer, New York, second edition, 1998.

[48] C. Marché. Normalized Rewriting: An Alternative to Rewriting Modulo a Set of Equations. *Journal of Symbolic Computation*, 11:1-36, 1996.

[49] Ralph N. McKenzie, George F. McNulty, Walter F. Taylor. *Algebras, Lattices, Varieties: Volume I*, Wadsworth & Brooks/Cole, Monterey, California, 1987.

[50] Maurice Mignotte, Doru Stefănescu. *Polynomials: An Algorithmic Approach*, Springer, New York, 1999. ISBN 981-4021-51-2.

[51] Theo Mora. Gröbner Bases for Non-commutative Polynomial Rings. In Jacques Calmet (Editor), *AAECC-3*, number 229 in Lecture Notes of Computer Science, page 353-362. Springer, Berlin, 1986.

[52] Arnold Neumaier. *Evaluation of Distributions.* Personal communication, May 2003.

[53] Günter F. Pilz. Algebra: Ein Reiseführer durch die schönsten Gebiete. Lecture notes, second edition,, Johannes Kepler University of Linz, Universitätsverlag Trauner, Linz, 1989.

[54] Fritz Reinhardt, Heinrich Soeder. *dtv Atlas zur Mathematik: Volume 1 and 2*, Deutscher Taschenbuch Verlag (dtv), München, seventh edition, 1987.

[55] John Alan Robinson. A Machine−oriented Logic Based on the Resolution Principle. *J. Assoc. Comp. Mach.*, 12(1):23-41, 1965.

[56] Markus Rosenkranz, Bruno Buchberger, Heinz W.Engl. *Computing the Moore−Penrose Inverse by Groebner Bases*. Conference on Computational Methods for Inverse Problems Strobl, Austria, August 27 2002.

[57] Markus Rosenkranz, Bruno Buchberger, Heinz W. Engl. Solving Linear Boundary Value Problems via Non-Commutative Gröbner Bases. *Applic. Anal.*, to appear:2003. Hardcopy number {6}.

[58] Markus Rosenkranz, Bruno Buchberger, Heinz W. Engl. Solving Linear Boundary Value Problems via Non-Commutative Gröbner Bases. In Koji Nakagawa (Editor), *Symposium in Honor of Bruno Buchberger's 60th Birthday: Logic, Mathematics and Computer Science: Interactions (LMCS 2002)*, page 217-230. Johannes Kepler University of Linz, Research Institute for Symbolic Computation, October 20-22, 2002. Proceedings by RISC-Linz, Technical Report No. 02-60, Castle of Hagenberg, Austria.

[59] Markus Rosenkranz, Bruno Buchberger, Heinz W. Engl. Solving Linear Boundary Value Problems via Non-Commutative Gröbner Bases. In Dana Petcu, Viorel Negru, Daniela Zaharie, Tudor Jebelean (Editors), *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'02)*, page 300-313. University of the West, Timisoara, Romania, October 9–12, 2002. Proceedings by Editura Mirton, Timisoara, Romania. Hardcopy number {7}.

[60] Markus Rosenkranz. Symbolic Solution of Simple BVPs on the Operator Level: A New Approach. *SIGSAM Bulletin*, to appear:2003.

[61] Joseph Shoenfield. *Mathematical Logic*, Addison-Wesley, Reading, Massachusetts (USA), 1967.

[62] J. Sebastião e Silva. Integrals and Orders of Growth of Distributions. In Centro de Calculo Cientifico (Editor), *Theory of Distributions (Proceedings of an International Summer Institute Held in Lisbon)*, pages 329-388. September 1964. Proceedings by Fundacao Calouste Gulbenkian, Lisbon. Hardcopy number {21}.

[63] Ivar Stakgold. *Green's Functions and Boundary Value Problems*, series Pure and Applied Mathematics. John Wiley & Sons, New York, 1979.

[64] Elena Tomuta. *An Architecture for Combining Provers and its Applications in the Theorema System*. PhD thesis, Research Institute for Symbolic Computation, Johannes Kepler University Linz, Castle of Hagenberg, Austria, July 1998.

[65] Wojciech A. Trybulec. Groups. *Journal of Formalized Mathematics (online)*, 2:1, 1990. See.

[66] V. Ufnarovski. Introduction to Noncommutative Gröbner Bases Theory. In Bruno Buchberger, Franz Winkler (Editors), *Gröbner Bases and Applications*, number 251 in London Mathematical Society Lecture Notes, page 259-280. Cambridge University Press, Cambridge (UK), 1998.

[67] John Wavrik. Rewrite Rules and Simplification of Matrix Expressions. *Computer Science Journal of Moldova*, 4(2/11):1996.

[68] Wolfgang Windsteiger. Building up Hierarchical Mathematical Domains Using Functors in Mathematica. In Alessandro Armando, Tudor Jebelean (Editors), *Systems for Integrated Computation and Deduction (Calculemus'99 Workshop in Trento, Italy, July 11-12)*, Electronic Notes in Theoretical Computer Science, page 83-101. 1999. Proceedings by Elsevier Science Publishers, Amsterdam.

[69] Wolfgang Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*. PhD Thesis, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, May 2001.

[70] Franz Winkler. *Polynomial Algorithms in Computer Algebra*, Springer, Wien, 1996.

[71] Doron Zeilberger. A holonomic systems approach to special functions identities. *Journal of Computational and Applied Mathematics*, 32:331-368, 1990. Hardcopy number {16}.

[72] Aspects of Generalized Inverses in Analysis and Regularization. In M. Zuhair Nashed (Editor), *Generalized Inverses and Applications: Proceedings of an Advanced Seminar Sponsored by the Mathematics Research Center*, page 193-244. University of Wisconsin-Madison (October 1973), 1976. Proceedings by Academic Press, New York.