Chapter II
Automated Reasoning

Tudor Jebelean
Bruno Buchberger, Temur Kutsia, Nikolaj Popov, Wolfgang Schreiner,
Wolfgang Windsteiger

Introduction 1

Observing is the process of obtaining new knowledge, expressed in language, by bringing the senses in contact with reality. Reasoning, in contrast, is the process of obtaining new knowledge from given knowledge, by applying certain general transformation rules that depend only on the *form* of the knowledge and can be done exclusively in the brain without involving the senses. Observation and reasoning, together, form the basis of the scientific method for explaining reality. Automated reasoning is the science of establishing methods that allow to replace human step-wise reasoning by procedures that perform individual reasoning steps mechanically and are able to find, automatically, suitable sequences of reasoning steps for deriving new knowledge from given one.

The importance of automatic reasoning originates in the fact that basically everything relevant to current science and technology can be expressed in the language of logic on which current automated reasoners work, for example the description of systems (specification or implementation of hardware and software), information provided on the internet, or any other kind of facts or data produced by the sciences. As the complexity of the knowledge produced by the observing sciences increases, the methods of automatic reasoning become more and more important, even indispensable, for mastering and developing our working and living environment by science and technology. In the same way as, over the millennia, humans developed tools for enhancing and amplifying their physical power and later developed tools (e.g. devices in physics) for enhancing the observing power, it is now the natural follow-up to develop tools for enhancing and amplifying the human reasoning power.

Mathematical Logic as Basis for Automated Reasoning

As Mathematics can be seen as the science of operating in abstract models of the reality (*thinking*), Mathematical Logic can be seen as the science of operating in abstract models of mathematical thinking (*thinking about thinking*). Since abstract models are expressed using statements, and operating in abstract models is done by transforming and combining these statements, Mathematical Logic studies their *syntax* (how do we construct statements), their *semantics* (what is the meaning of statements) and their *pragmatics* (rules that describe how statements can be transformed in a way that respects semantics). In the era of electronic computing, the importance of automated reasoning increases tremendously, because computers are devices for automatic operation in abstract models (*thinking tools*). Thus, Mathematical Logic becomes also the theoretical basis for studying the design and the behavior of computing devices and programs and, hence, Automated Reasoning is *automated thinking about thinking tools*.

Automated Mathematical Theorem Proving

Since logical formulae have been traditionally used for expressing mathematics, there is a widespread opinion that automated reasoning can be used only for proving mathematical statements, which is sometimes perceived as either redundant (in case of already proven theorems) or hopeless (in case of not yet proven conjectures). First let us emphasize that automated mathematical theorem proving is only a part of automated reasoning—however crucial because it develops techniques which are useful in all other areas of science and technology. Moreover automatic theorem proving is neither redundant (because proving "known" or "trivial" theorems is absolutely necessary in the process of [semi]-automatic verification of complex systems—hardware, software, or combined hardware/software), nor hopeless (because, on the other hand, the proofs of highly nontrivial theorems as the *Four Color Theorem* [AH77] and the *Robbins Conjecture* [McC97] were only possible by the use of automatic theorem proving tools).

Verification and Synthesis

Contemporary technological systems consist of increasingly complex combinations of hardware, software, and human agents, whose tasks are very sophisticated. How do we *express* these sophisticated tasks, how do we *design* and how do we *describe* these technological systems, and how do we *ensure* that the systems always fulfill their tasks? Those who believe that (at least in some organization with a long technological tradition) *these four questions* have been properly answered may take a look at some famous software failures `http://en.wikipedia.org/wiki/List_of_notable_software_bugs`.

The consequences of design defects in complex technological systems have become a part of our everyday life: computer viruses, unauthorized access to sensitive data (e. g. bank accounts and credit cards), and periodic failures of the programs on our computers and on our mobile phones. The future brings: automotive software for handling the controls and the airbags in our automobiles, generalized internet banking, and the inclusion of computers in most of the objects around us.

Today it is largely accepted that the answer to the above four questions is: both the description of the complex systems (*implementations*), as well as their sophisticated tasks (*specifications*) can be expressed as logical formulae, the design of complex systems can be decomposed in successive and controllable steps of transformation of such logical formulae, and the verification of their correct behavior can be performed by checkable inferences on these formulae.

Semantic Representation of the Information on the Internet

The extraordinary proliferation of the data which is accessible on the internet offers of course an unprecedented richness of information at our fingertips, however the limitations of the current *syntactic* approach are more and more visible. It is often very difficult for the user to select the relevant information among the "noise" of irrelevant one, and it is also impossible to find out pieces of knowledge which require a minimal amount of intelligent processing. These problems can be solved only by a *semantic* approach: the information has to be stored in form of logical statements (probably of very simple structure, but high quantity), and the search engines have to include Automatic Reasoning capabilities.

This chapter summarizes the work performed in the Softwarepark Hagenberg in the field of Automated Reasoning, in particular the work performed at RISC and in the *Theorema* group. Research from other groups in Hagenberg are also tangent with Automated Reasoning, and they are mentioned in the respective chapters.

2 *Theorema*: Computer-Supported Mathematical Theory Exploration

At RISC, much of the research on automated reasoning focuses on the *Theorema* Project, which aims at developing algorithmic methods and software tools for supporting the intellectual process of *mathematical theory exploration.* The emphasis of the *Theorema* Project is not so much on the automated proof of yet unknown or difficult theorems but much more on organiz-

ing the overall flow of the many small reasoning steps necessary in building up mathematical theories or writing proof-checked mathematical textbooks and lecture notes or developing verified software. The net effect of an exploration, however, may also be that complicated theorems and nontrivial algorithms can be proven correct with only very little user-interaction necessary at some crucial stages in the exploration process, while the individual intermediate reasoning steps are completely automatic. An example of a non-trivial automated algorithm synthesis (the synthesis of a Gröbner bases algorithm) by the *Theorema* methodology is given later in this chapter. The main contribution of the working mathematician who uses *Theorema* will then be the organization of a well structured exploration process that leads from the initial knowledge base to the full-fledged theory.

This design principle of *Theorema* is in contrast to the main stream in automated mathematical theorem proving, which to a great extent has focused on proving individual theorems from given knowledge bases (containing the axioms of the theory, definitions, lemmata etc.). Considering the mathematical theory exploration process (invention of *notions*, invention and proof/refutation of *propositions*, invention of *problems*, invention and verification of *algorithms/methods* for solving problems) and the computer-supported documentation of this process as a coherent process seems to be more natural and useful for the success of automated theorem proving for the every-day practice of working mathematicians than considering the proof of isolated theorems. This point of view has been made explicit, first, in [Buc99] and, later, in [Buc03, Buc06].

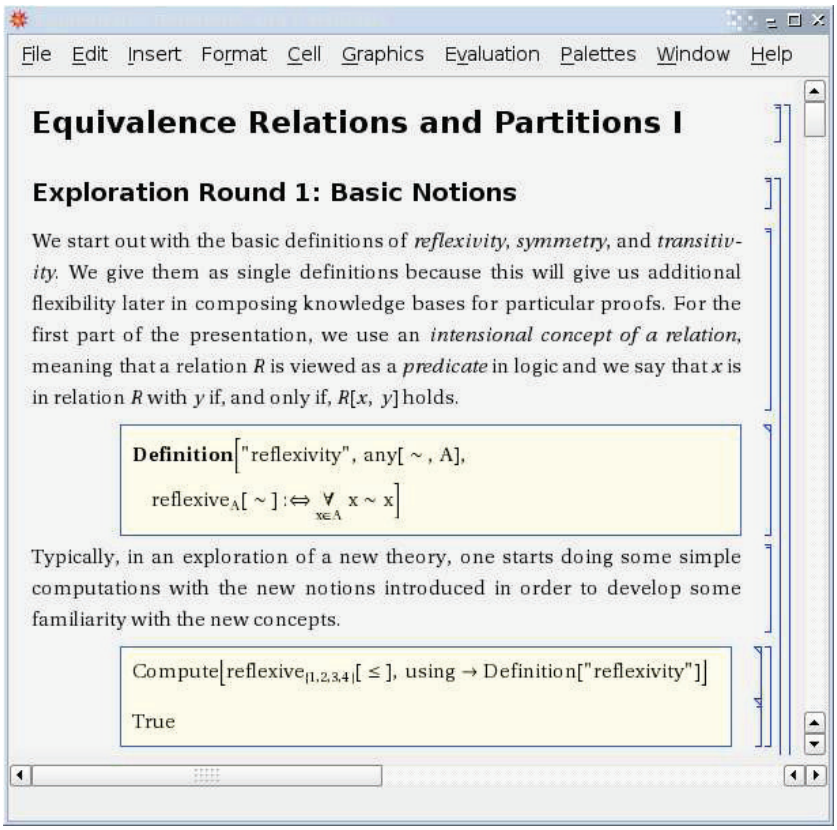
The *Theorema* Group has strived to contribute, in various ways, to the computer-support of the mathematical theory exploration process by building

- tools for the automated generation of proofs in various general theories (e.g. elementary analysis, geometry, inductive domains including natural number theory and tuple theory, and set theory),
- and tools for the organization of the theory exploration process and build-up of mathematical knowledge bases (various viewers for proofs including the "focus window" approach, proof presentation including natural language explanation, logico-graphic symbols, user-defined two-dimensional syntax, functors for domain building etc.).

The research goals and the basic design principles of the *Theorema* project were formulated in a couple of early papers, see [Buc96b, Buc96a, Buc96c, Buc97]. Summaries of the achievements in the *Theorema* Project are [BJK+97, BDJ+00, BCJ+06]. A complete list of the publications of the *Theorema* Group can be accessed on-line at `www.theorema.org`.

The *Theorema* Language and the User Interface 2.1

The typical user interface of *Theorema* is the Mathematica notebook frontend, which allows the combination of mathematical formulae and natural language text (and much more) in a natural way. Figure 1 shows a screenshot of a typical *Theorema* notebook that exhibits the main components of the *Theorema* language. An important design principle of the *Theorema* sys-



Theorema input in a Mathematica notebook. FIGURE 1

tem is to come as close to conventional mathematics in the appearance of mathematics in as many aspects as possible, be it in the language in which mathematics is expressed, be it in the way how proofs are presented, and many more.

The *Theorema* language is structured in essentially three layers,

- the *Theorema* formula language,
- the *Theorema* formal text language, and
- the *Theorema* command language.

These three layers correspond to three aspects of mathematical language, namely the logical part of *formulating statements* in a concise and correct way, the organizational part of *structuring knowledge* into definitions, theorems, lemmata, etc., or entire theories, and the description of various *mathematical activities* like proving or computing.

The *Theorema* formula language is a version of higher order predicate logic without extensionality. On this basis, the language offers sets, tuples, and certain types of numbers as basic language components. As an example,

$$\underset{x\in A}{\forall}\ x \sim x \tag{1}$$

is a statement in the *Theorema* formula language. As can be seen in Figure 1, *Theorema* allows standard mathematical notation even in input so that formulae can be written like in conventional mathematical texts. Two-dimensional input including also special symbols (like integrals or quantifiers) is standard technology in Mathematica. In order to facilitate *Theorema* input, we provide specialized palettes that paste skeletons for frequently used *Theorema* expressions into a notebook by just one mouse-click.

For composing and manipulating large formal mathematical texts, however, we need to be able to combine the expression language with auxiliary text (labels, key words like "Definition", "Theorem", etc.) and to compose, in a hierarchical way, large mathematical knowledge bases from individual expressions. In the example, in order to define reflexivity by (1) we would use a definition environment.

Definition["reflexivity", any$[A, \sim]$,

$\text{reflexive}_A[\sim] :\Longleftrightarrow \underset{x\in A}{\forall}\ x \sim x$ "$\sim\circlearrowleft$"]

The field "any[*vars*]" declares *vars* as (the free) variables. Each variable v in *vars* can carry a type restriction of the form "$v \in S$" or "type[v]". An optional field "with[*cond*]" tells that the variables must satisfy the condition *cond*. Logically, the variable declaration and the condition are just a shortcut for prefixing every formula in the environment with a universal quantifier. Other examples of formal text are:

Definition["class", any$[x, A, \sim]$, with$[x \in A]$,

$\text{class}_{A,\sim}[x] := \{a \in A \mid a \sim x\}$]

Lemma["non-empty class", any$[x \in A, A, \sim]$, with[$\text{reflexive}_A[\sim]$],

$\text{class}_{A,\sim}[x] \neq \emptyset$]

The effect of entering an environment into the system is that its content can be referred to later by *Keyword*[*env_label*]. Knowledge can be grouped

using *nested environments*, whose structure is identical except that instead of clauses (formulae with optional labels) there are references to previously defined environments. Typical keywords used for nested environments are "Theory" and "KnowledgeBase", e.g.

Theory["relations",
  Definition["reflexivity"]
  Definition["class"]]

The mathematical activities that are supported in the command language are computing, proving, and solving. Computations are performed based on a semantics of the language expressed in the form of computational rules for the finitary fragment of the formula language, i.e. finite sets, tuples, numbers, and all sorts of quantifiers as long as they involve a finite range for their variable(s). In the example,

`Compute`[class$_{\{1,2,3,4\},\leq}$[3], using→Definition["class"],
  built-in→ {Built-in["Sets"], Built-in["Numbers"]}]

would compute the class of 3 in $\{1, 2, 3, 4\}$ w.r.t. $\leq$ using the definition of class (see above) and built-in semantics of (finite) sets and numbers resulting in $\{1, 2, 3\}$ and

`Compute`[reflexive$_{\{1,2,3,4\}}$[$\leq$], using→Definition["reflexivity"],
  built-in→ {Built-in["Quantifiers"], Built-in["Numbers"]}]

would decide by a finite computation, whether the relation $\leq$ is reflexive on $\{1, 2, 3, 4\}$ using the definition of reflexivity and the built-in semantics of quantifiers and numbers resulting in "true". Consider the lemma about non-empty classes stated above, which is a statement about relations on arbitrary not necessarily finite sets. Thus, its validity cannot be verified by computation but must be proven. In order to prove a statement in *Theorema*, we use

`Prove`[Lemma["non-empty class"], using→Theory["relations"],
  by→SetTheoryPCSProver],

which will try to prove Lemma["non-empty class"] using Theory["relations"] as the knowledge base by SetTheoryPCSProver, a prove method for set theory described in more detail in Section 3.2. In case of success, the complete proof is presented in human readable nicely structured format in a separate window, otherwise the failing proof attempt is displayed. Moreover, *Theorema* features a novel approach for displaying proofs based on focus windows [PB02], a proof simplification tool, and an interactive proof tool [PK05].

2.2 "Lazy Thinking": Invention by Formulae Schemes and Failing Proof Analysis

A main point in the *Theorema* approach to mathematical theory exploration is that mathematical invention should be supported both "bottom-up", by using formulae schemes, and "top-down", by analyzing failing proofs and constructing guesses for necessary intermediate lemmata. This combined approach is called "lazy thinking" and was introduced in [Buc03].

The difficulty of finding proofs for propositions depends, to a large extent, on the available knowledge. Most mathematical inventions, even simple ones like the proof of, say, the lemma that the limit of the sum of two sequences is equal to the sum of the limits, would hardly be possible (even for quite intelligent humans) if mathematical theories were not built up in small steps. In each step, only one new concept is introduced (by an axiom or definition) and all possible simple knowledge is proved first before the proof of any more important theorem is attacked. With sufficiently much intermediate knowledge, it often turns out that the proof of the essential theorems then only needs one single or very few "difficult" ideas that cannot be generated completely automatically.

It is rewarding to scrutinize on what typically happens in a step in which propositions for new notions are conjectured: In fact, in most cases, the type of knowledge conjectured has "rewrite" character: For example, if the notion of multiplication on natural numbers has been introduced, then all possible interactions of this new notion with previous notions like 'zero', 'addition', 'less' etc. that can be formulated as "rewrite properties" should be studied first. For example, distributivity is such a property in rewrite form:

$$(x + y) * z = x * z + y * z.$$

It is an important observation that, when sufficiently many rewrite properties have been proven by using the "fundamental" (sometimes difficult) proof methods in the theory, subsequent proofs of most other possible properties then can be done by simple "rewrite proving" ("symbolic computation proving", "physicists proving", "quantifier free proving", "highschool proving"), i.e. by applying the proven rewrite properties repeatedly just using substitution and replacement. (In the theory of natural numbers, the "fundamental" proving method is induction; in elementary analysis, the "fundamental" proving method is general predicate logic for "alternating" quantifiers '$\forall\ \exists$'; etc.). A good theory exploration environment, should support this important observation. In the *Theorema* Project, this observation is a guiding strategy.

How can (rewrite and other) knowledge about notions (introduced by definitions) be "invented", i.e. systematically generated? In the "lazy thinking" approach introduced in [Buc03], two complementary strategies are proposed:

1. The use of "formulae schemes", a bottom-up approach.
2. The use of "analysis of failing proofs", a top-down approach.

A synopsis of the lazy thinking approch to the automation of mathematical theory exploration and some more details can also be found, for example, in [Buc06].

The lazy thinking strategy can be applied both to the invention and verification of theorems and the invention and verification of algorithms ("algorithm synthesis"). Here, we illustrate the method by two examples of algorithm synthesis. There is a rich literature on algorithm synthesis methods, see the survey [BDF+04]. Our method, in the classification given in this survey, is in the class of "scheme-based" methods but is essentially different from previously known such methods by its emphasis on the heuristic usefulness of *failing* correctness proofs.

The algorithm synthesis problem is the following problem: Given a problem specification P (i.e. a binary predicate $P[x, y]$ that specifies the relation between the input values x and the output values y of the problem), find an algorithm A such that

$$\underset{x}{\forall}\, P[x, A[x]].$$

The lazy thinking approach to algorithm synthesis consists of the following steps:

- Consider known fundamental ideas ("*algorithm schemes*") of how to structure algorithms A in terms of sub-algorithms B, C, ... Try one scheme A after the other.
- For the chosen scheme A, try to prove $\underset{x}{\forall}\, P[x, A[x]]$. This proof will probably fail because, at this stage, nothing is known about the sub-algorithms B, C, ... From the *failing proof, construct specifications* Q, R, ... for the sub-algorithms B, C, ... that make the proof work.
- Then A together with any sample of algorithms B, C, ... that satisfy the specifications Q, R, ... will be a correct algorithm for the original problem P.
- If such sub-algorithms B, C, ... are available in the given knowledge base, then we are done, i.e. an algorithm for problem P has been synthesized. If no such algorithms are available, we can apply the lazy thinking method, recursively, for synthesizing algorithms B, C, ... that satisfy Q, R, ... until we arrive at specifications that are met by available algorithms in the knowledge base.

For the (automated) construction of specifications from failing correctness proofs we introduced the following simple (but amazingly powerful) rule: In the failing correctness proof, collect the temporary assumptions

$$T[x_0, \ldots, A[\ldots], \ldots]$$

(where $x_0, \ldots$ are the constants resulting from the "arbitrary but fixed" proof rule) and the temporary goals

$$G[x_0, \ldots, B[\ldots, A[\ldots], \ldots]]$$

and produce the specification for sub-algorithm B:

$$\underset{X,Y,\ldots}{\forall} T[X,\ldots,Y,\ldots] \implies G[Y,\ldots,B[\ldots,Y,\ldots]].$$

We illustrate the method in a simple example: We synthesize, completely automatically, an algorithm for the sorting problem, which is the problem to find an algorithm A such that

$$\underset{X}{\forall} \text{is-sorted-version}[X, A[X]].$$

We assume that the binary predicate 'is-sorted-version' is defined by a set of formulae in predicate logic. In the first step of the lazy thinking approach, we choose one of the many algorithm schemes in our library of algorithm schemes, for example, the 'Divide-and-Conquer' scheme, which can be defined, within predicate logic, by

$$\underset{A,S,M,L,R}{\forall} \text{Divide-and-Conquer}[A,S,M,L,R] \iff$$
$$\underset{x}{\forall} A[x] = \begin{cases} S[x] & \Leftarrow \text{is-trivial-tuple}[x] \\ M[A[L[x]], A[R[x]]] & \Leftarrow \text{otherwise} \end{cases}$$

This is a scheme that explains how the unknown algorithm A should be defined in terms of unknown subalgorithms S, M, L, R. With this knowledge we try to prove that

$$\underset{X}{\forall} \text{is-sorted-version}[X, A[X]]$$

using one of our automated provers (for induction over tuples). This proof will fail because, at this moment, nothing is known about the subalgorithms S, M, L, R. Anaylizing the failing proof for the pending goals and available temporary knowledge at the time of failure we now use the above rule for generating, automatically, specifications for S, M, L, R that will make the proof work. In this example, in approx. 2 minutes on a laptop, the following specifications are generated automatically:

$$\underset{x}{\forall} \text{is-trivial-tuple}[x] \implies S[x] = x,$$
$$\underset{y,z}{\forall} \begin{matrix} \text{is-sorted}[y] \\ \text{is-sorted}[z] \end{matrix} \implies \begin{matrix} \text{is-sorted}[M[y,z]] \\ M[y,z] \approx (y \asymp z) \end{matrix},$$
$$\underset{x}{\forall} L[x] \asymp R[x] \approx x.$$

(Here, '$\asymp$', and '$\approx$' denote "concatenation" and "equivalence" of tuples.) A closer look to the formulae reveals the amazing fact that these specifications on S, M, L, R are not only sufficient for guaranteeing the correctness of A

but are also completely natural and intuitive: They tell us that a suitable algorithm S must essentially be the identity function, suitable algorithms L and R must essentially be "pairing functions" (which split a given tuple X in two parts that, together, preserve the entire information in X) and that M must be a merging algorithm.

Automated Synthesis of Gröbner Bases Theory

Our expectation was that, with lazy thinking, one may be able to synthesize only quite simple algorithms. It came as a surprise, see [Buc04], that, in fact, algorithms for quite non-trivial problems can be synthesized by this method. The most interesting example so far is the problem of Gröbner bases construction with the specification: Find an algorithm Gb, such that

$$\underset{\text{is-finite}[F]}{\forall} \begin{array}{l} \text{is-finite}[\text{Gb}[F]] \\ \text{is-Gröbner-basis}[\text{Gb}[F]] \\ \text{ideal}[F] = \text{ideal}[\text{Gb}[F]] \end{array}.$$

(The quantifier ranges over sets F of multivariate polynomials. 'ideal$[F]$' is the set of all linear combinations of polynomials from F.) In Chapter I on symbolic computation it is explained why this problem is non-trivial and why it is important and interesting. In fact, the problem was open for over 60 years before it was solved in [Buc65]. Thus, it may be philosophically and practically interesting that now it can be solved automatically, i.e. the key idea of algorithmic Gröbner bases theory, namely the notion of S-polynomials, and the algorithm based on this key idea can be generated automatically from the specification of the problem by the lazy thinking method.

Namely, we start with the following algorithm scheme, called "*Pair Completion*", that tells us that the unknown algorithm Gb should be defined in terms of two unknown subalgorithms lc and df in the following way:

$$\begin{array}{l} \underset{\text{Gb,lc,df}}{\forall} \text{Pair-Completion}[\text{Gb}, \text{lc}, \text{df}] \iff \\ \quad \underset{F}{\forall}\ \text{Gb}[F] = \text{Gb}[F, \text{pairs}[F]] \\ \quad \underset{F}{\forall}\ \text{Gb}[F, \langle\rangle] = F \\ \quad \underset{F, g_1, g_2, \bar{p}}{\forall}\ \text{Gb}[F, \langle\langle g_1, g_2\rangle, \bar{p}\rangle] = \\ \quad \text{where}[f = \text{lc}[g_1, g_2], h_1 = \text{trd}[\text{rd}[f, g_1], F], h_2 = \text{trd}[\text{rd}[f, g_2], F], \\ \quad \left\{ \begin{array}{ll} \text{Gb}\,[F, \langle\bar{p}\rangle] & \Leftarrow h_1 = h_2 \\ \text{Gb}[F \frown \text{df}[h_1, h_2], \langle\bar{p}\rangle \asymp \langle\langle F_k, \text{df}[h_1, h_2]\rangle \underset{k=1,\ldots,\lvert F\rvert}{\mid} \rangle] & \Leftarrow \text{otherwise} \end{array} \right] \end{array}$$

(Here, our notation for tuples is '$\langle \ldots \rangle$' and '$\frown$' is the append function. The function 'rd' is the one-step reduction function and the function 'trd' is total reduction, i.e. the iteration of 'rd' until an irreducible element is reached.) Now we attempt to prove, automatically, that the above specification holds for the algorithm Gb that is defined in this way from unknown algorithms lc and df. An automatic prover that is powerful enough for this type of proof was implemented in [Cra08]. The proof fails because, at this stage, nothing is known about lc and df. Using the above specification generation rule, one can generate, completely automatically, the following specification for lc.

$$\underset{p,g_1,g_2}{\forall} \begin{matrix} \mathrm{lp}[g_1]|p \\ \mathrm{lp}[g_2]|p \end{matrix} \Longrightarrow \mathrm{lc}[g_1, g_2]|p,$$
$$\mathrm{lp}[g_1]|\mathrm{lc}[g_1, g_2],$$
$$\mathrm{lp}[g_2]|\mathrm{lc}[g_1, g_2],$$

which shows that a suitable subalgorithm lc is essentially the least common multiple of the leading power products of the polynomials g_1 and g_2. Similarly one automatically obtains that df must essentially be the difference of polynomials. These two ideas are the main ingredients of the notion of S-polynomials, which is in fact the main idea of algorithmic Gröbner bases theory (see Chapter I on symbolic computation). This idea, together with its correctness proof, comes out here completely automatically. This is currently one of the strongest results of the *Theorema* project which creates quite some promises for the future of semi-automated mathematical theory exploration.

3 Natural Style Proving in *Theorema*

The *Theorema* system contains several provers, which differ both in their methods and in the domains which are treated. However, all *Theorema* provers work in *natural style*, that is: the proofs are presented in natural language, and the proof structure and the logical inferences are similar to the ones used by humans. Moreover, in the context of the *Theorema* system one may use provers which have implicit knowledge about the used domain (e.g. number domains), like for instance the PCS prover. This makes certain proofs more compact and readable, in contrast to proving in pure predicate logic with explicit assumptions for such theories.

In this section we summarize shortly the provers of the *Theorema* system, and then we focus on two particular provers: the S-decomposition prover and the set theory prover. All provers are presented in more detail in our survey

papers [BJK+97, BDJ+00, BCJ+06] and in the publications available on our home page `www.theorema.org`.

The provers available in *Theorema* include: a general predicate logic prover, various induction provers containing a simple rewrite prover as a component, a special prover for proving properties of domains generated by functors, the PCS prover for analysis (and similar theories that involve concepts defined by using alternating quantifiers) [Buc01], a set theory prover (using the PCS approach as a subpart), a special prover for geometric theorems using the Gröbner bases method [Rob02], a special prover for combinatorics using the Zeilberger–Paule approach, the cascade mechanism for inventing lemmata on the way to proving theorems by induction, an equational prover based on Knuth-Bendix completion [Kut03], and a basic reasoner [WBR06].

3.1 S-Decomposition and the Use of Algebraic Techniques

Numerous interesting mathematical notions are defined by formulae that contain a sequence of "alternating quantifiers", i.e., the definitions have the structure $p[x, y] \Leftrightarrow \underset{a}{\forall}\, \underset{b}{\exists}\, \underset{c}{\forall} \ldots q[x, y, a, b, c]$. Many notions introduced, for example, in elementary analysis text books (limit, continuity, function growth order, etc.) fall into this class. Therefore, it is highly desirable that mathematical assistant systems support the exploration of theories about such notions.

The S-decomposition method is particularly suitable both for proving theorems (when the auxiliary knowledge is rich enough) as well as conjecturing propositions (similar to Lazy Thinking) during the exploration of theories about notions with alternating quantifiers. It can be seen as a further refinement of the Prove-Compute-Solve method implemented in the *Theorema* PCS prover [Buc01]. Essentially, the S-decomposition method is a certain strategy for decomposing the proof into simpler subproofs, based on the structure of the main definition involved. The method proceeds recursively on a group of assumptions together with the quantified goal, until the quantifiers are eliminated, and produces some auxiliary lemmata as subgoals.

We present the method using an example from elementary analysis: limit of a sum of sequences; see [Jeb01] for a detailed description of the method. The definition of "f converges to a" is:

$$(\rightarrow) \quad f \rightarrow a \;\Leftrightarrow\; \underset{\epsilon}{\forall} \left(\epsilon > 0 \Rightarrow \underset{m}{\exists}\, \underset{n}{\forall} \left(n \geq m \Rightarrow |f[n] - a| < \epsilon\right)\right).$$

(For brevity, the type information is not included.)

The proof tree is presented in Figure 2 and Figure 3. Boxes represent proof situations (with the goal on top), unboxed formulae represent auxiliary subgoals, and boxes with double sidebars represent substitutions for the

metavariables. The nodes of the proof tree are labeled in the order they are produced.

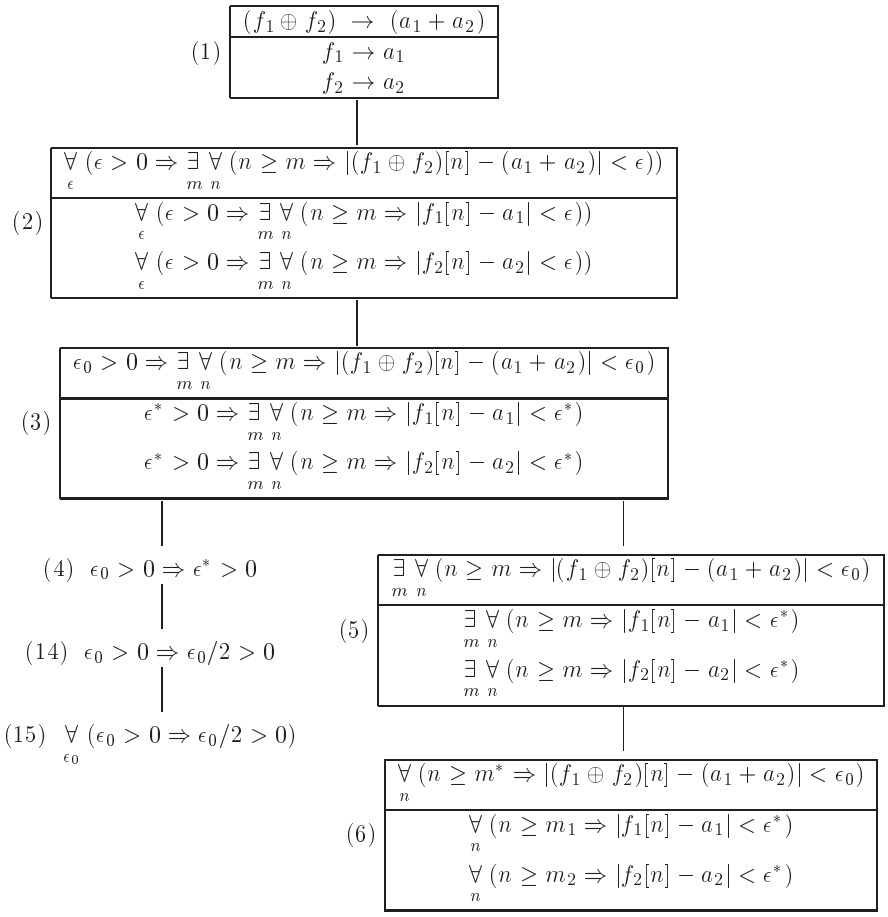
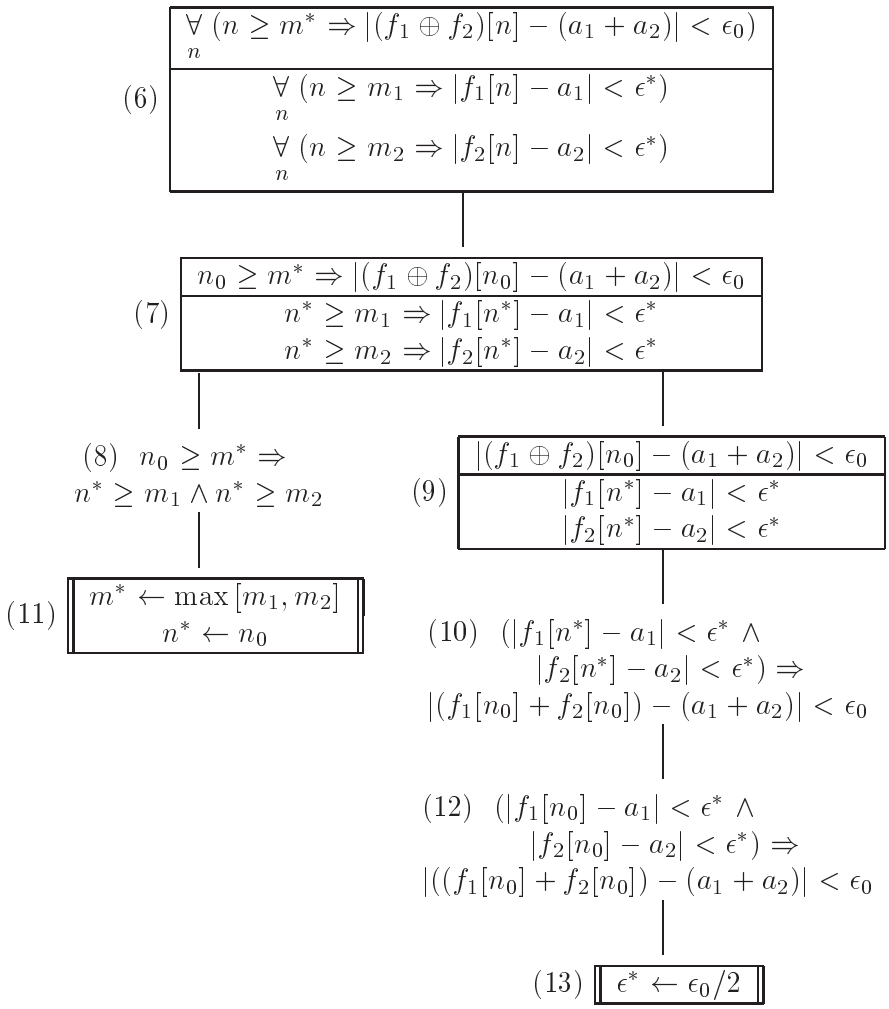


FIGURE 2 S-Decomposition: First part of the proof tree.

The first inference expands the definition of "limit", generating the proof situation (2). *S-decomposition is designed for proof situations in which the goal and the main assumptions have exactly the same structure.* In the example they differ only in the instantiations of f and a. *S-decomposition proceeds by modifying these formulae together, such that the similarity of the structure is preserved, until all the quantifiers and logical connectives are eliminated.* The method is specified as a collection of four transformation rules (infer-



S-Decomposition: Second part of the proof tree. FIGURE 3

ences) for proof situations and a rule for composing auxiliary lemmata. The transformation rules are described below together with their concrete application to this particular proof.

The inference that transforms (2) to (3) eliminates the universal quantifier and has the general formulation below. (Here, for simplicity, we formulate the inferences for two assumptions only, but extending them to use an arbitrary number of assumptions is straightforward.)

$$\underset{x}{\forall} P_1[x], \underset{x}{\forall} P_2[x] \;\vdash\; \underset{x}{\forall} P_0[x] \;\longmapsto\; P_1[x_1^*], P_2[x_2^*] \;\vdash\; P_0[x_0] \qquad (\forall)$$

Like the existential rule, specified later in this section, this rule combines the well-known techniques for introducing Skolem constants and metavariables. However, S-decomposition comes with a *strategy* of applying them in a certain order. The Skolem constant x_0 is introduced before the metavariables (names for yet unknown terms) x_1^*, x_2^*. In the example we use a simplified version of this rule in which the metavariables do not differ. For other examples (e.g. quotient of sequences) this will not work.

The inference from (3) to (4) and (5) eliminates the implication, and has the general formulation:

$$Q_1 \Rightarrow P_1,\; Q_2 \Rightarrow P_2 \;\vdash\; Q_0 \Rightarrow P_0 \;\longmapsto\; \begin{cases} Q_0 \Rightarrow Q_1 \wedge Q_2 \\ P_1, P_2 \;\vdash\; P_0 \end{cases} \qquad (\Rightarrow)$$

In contrast to the previous rule, this one is not an equivalence transformation (the proof of the right-hand side might fail even if the left-hand side is provable). This rule is applied in the situations when Q_k's are the "conditions" associated with a universal quantifier (as in the example). The formula $Q_0 \Rightarrow Q_1 \wedge Q_2$ is a candidate for an auxiliary lemma, as is formula (4).

The proof proceeds further with the transformation (5)–(6) (formula (14) will be produced later in the proof) given by the following rule:

$$\underset{x}{\exists} P_1[x], \underset{x}{\exists} P_2[x] \;\vdash\; \underset{x}{\exists} P_0[x] \;\longmapsto\; P_1[x_1], P_2[x_2] \;\vdash\; P_0[x^*] \qquad (\exists)$$

where x_1 and x_2 are Skolem constants introduced before the metavariable x^*.

Usually, existential quantifiers are associated with conditions upon the quantified variables. In such a case one would obtain conjunctions (analogous to the situation in formula (3), where one obtains implications). The rule for decomposing conjunctions is:

$$Q_1 \wedge P_1,\; Q_2 \wedge P_2 \;\vdash\; Q_0 \wedge P_0 \;\longmapsto\; \begin{cases} Q_1 \wedge Q_2 \Rightarrow Q_0 \\ P_1, P_2 \;\vdash\; P_0 \end{cases} \qquad (\wedge)$$

Similarly to the rule ($\Rightarrow$), this rule produces an auxiliary lemma as a "side effect", using the Q_k's which are, typically, the conditions associated with an existential quantifier. In fact, in the implementation of the method, the rules ($\exists$), ($\wedge$) are applied in one step, as are also the rules ($\forall$), ($\Rightarrow$).

However, in this example there is no condition associated to the existential quantifier, therefore this rule is not used.

The proof proceeds by applying rule ($\forall$) to (6), and then the rule ($\Rightarrow$) to (7). Note that the transformation rules proceed from the assumptions towards the goal for existential formulae, and the other way around for universal formulae. If one would illustrate this process by drawing a line on the formulae

in proof situation (2), one obtains an S-shaped curve—thus the name of the method.

Finally, S-decomposition transforms a proof situation having no quantifiers into an implication, thus (9) is transformed into (10), and this finishes the application of S-decomposition to this example. In this moment the original proof situation is decomposed into the formulae (4), (8), and (10). (Obtaining (10) needs an additional inference step, not shown in the figure, which consists in expanding the subterm $(f_1 \oplus f_2)[n_0]$ by the definition of $\oplus$.)

The continuation of the proof is outside the scope of the S-decomposition method. For completing the proof, one needs to find appropriate substitutions for the metavariables, such that the Skolem constants used in each binding are introduced earlier than the corresponding metavariable. For the sake of completeness, we give here a possible follow up (produced automatically by *Theorema*): We assume that the formulae

$$(21) \quad \underset{k,i,j}{\forall} (k \geq \max[i,j] \Rightarrow k \geq i \wedge k \geq j),$$

$$(22) \quad \underset{x,y,a,b,\epsilon}{\forall} \left(|x-a| < \frac{\epsilon}{2} \wedge |y-b| < \frac{\epsilon}{2} \Rightarrow |(x+y)-(a+b)| < \epsilon\right)$$

are present in the available knowledge as auxiliary assumptions. The prover first tries to "solve" (8), and by matching against (21) obtains the substitution (11). This substitution is applied to (10) producing (12), and by matching the latter against (22), the prover obtains the substitution (13). The substitutions are then applied to the formula (4), which is then generalized (by universal quantification of the Skolem constants) into (15). The latter is presented to the user as suggestions for auxiliary lemmata needed for completing the proof. Of course this subgoal would be also solved if the appropriate assumption was available, however the situation described above demonstrates that the method is also useful for generating conjectures.

The reader may notice that the process of guessing the right order in which the subgoals (4), (8), and (10) should be solved is nondeterministic and may involve some backtracking. This search is implemented in *Theorema* using the principles described in [KJ00].

Moreover, this method can be used in conjunction with algebraic techniques, in particular with Cylindrical Algebraic Decomposition [Col75]. Namely, the substitutions for the metavariables shown at steps (11) and (13) can be also obtained by using CAD–based quantifier elimination. First the proof situations (11) and (12) are transformed into quantified formulae: the metavariables become existential variables, the Skolem constants become universal variables, and the order of the quantifiers is the order in which the respective metavariables and Skolem constants have been introduced during S-decomposition. Then, by successive applications of quantifier elimination, one obtains automatically the witness terms for the existential variables. The method and its application to several examples are described in detail in [VJB08].

3.2 The *Theorema* Set Theory Prover

Many areas of mathematics are typically formulated on the basis of set theory, in the sense that objects or properties are expressed in terms of language constructs from set theory. Most prominently, set formations like

$$\{x \in A \mid P_x\} \quad \text{or} \quad \{T_x \mid x \in A\} \tag{2}$$

occur routinely in virtually all of mathematics. The *Theorema* language described in Section 2.1 supports all commonly used constructs from set theory, such as set formation as shown in (2), membership, union, intersection, power set, and many more. The semantics of the language built into the system immediately allows computations on *finite sets* including also the computation of truth values for statements containing finite sets. Reasoning on *arbitrary sets*, however, amounts to the application of more powerful techniques. This was the starting point for the development of a *set theory prover*, see [Win06] and [Win01], based on the general principles of "PCS" (Proving–Computing–Solving) reasoners introduced in [Buc01] in the frame of the *Theorema* system.

Integration of Proving and Computing

One of the design goals of this prover was the smooth integration of proving, i.e. general reasoning based on inference rules, and computing on numbers, finite sets, tuples and the like. In order to accomplish this task, the set theory prover contains a component that allows to apply computational rules defined in the semantics of the *Theorema* language to formulae occuring in a proof. By this mechanism, the user can even choose, which parts of the language semantics to include in a particular proof.

We demonstrate this in a simple example from a fully mechanized proof of the irrationality of $\sqrt{2}$ taken from a comparison of automated theorem provers carried out by Freek Wiedijk in 2006, see [WBR06]. During the formalization of this proof, one arrives at a formula

$$2m_0^2 = (2m_1)^2, \tag{3}$$

which of course simplifies by simple computation on natural numbers to

$$m_0^2 = 2m_1^2. \tag{4}$$

Compared to other systems, where either

- additional theorems are required to perform the step from (3) to (4)—and consequently separate theorems for all situations similar to this—or
- the simplification from (3) to (4) is carried out by a lengthy sequence of transformation steps based on the axioms for natural numbers,

the step simplifying (3) into (4) is only *one elementary step* based on the semantics of the natural numbers built-into the *Theorema* system. The proofs generated in this way are very elegant and close to how a human would give the proofs—one of the main credos in the design of the *Theorema* system.

The Theoretical Foundations of the Prover

One of the first questions when it comes to set theory is always: "How are the well-known contradictions appearing in naive set theory, e.g. Russell's paradox, avoided?" The *Theorema* set theory prover relies on the Zermelo-Frankel axiomatization of set theory (ZF), meaning that the prover can deal with all sorts of sets whose existence is guaranteed by the Zermelo-Frankel axioms for set theory. This means, in particular, that the *Theorema* language does not forbid "sets" like $\{x \mid x \notin x\} =: R$ nor does it forbid statements like $R \in R$. Rather, the set theory prover refuses to apply any inference step on $R \in R$ on the grounds that R is not formed by any of the set construction principles proven to be consistent in ZF—note that ZF requires $\{x \in S \mid P_x\}$ for some known set S when abstracting a set from a property P_x.

In addition to inference rules based directly on some ZF-axiom, e.g. the inference rule for membership in a set $\{x \in S \mid P_x\}$, the prover also incorporates knowledge *derivable* in ZF. If the prover was intended to be used to prove theorems of set theory based on the ZF axiomatization, it would be cheating if the prover has such knowledge already built in, hence, there is a mechanism to switch off these special rules in case a user wants to use the prover for this purpose. The main field of application for the prover is, however, to prove arbitrary statements whose formalization *uses* language constructs from set theory. An example of such a proof is shown in detail in Figure 4. This is an example of the TPTP library (SET722) of examples for automated theorem provers, and it says that if the composition of functions $g \circ f$ is surjective then also g must be surjective. Note that the knowledge base for this proof only contains the definition of composition, we need not give the definition of surjectivity, because this is built into the prover as a standard concept in set theory. Of course, the proof would also succeed with one's own definition of surjectivity in the knowledge base. The important difference lies in the concise proof produced by this prover because several elementary logical steps are combined into one step when the built-in rule is applied. Note also, that the proof generated by the *Theorema* system comes

out exactly as it is displayed in Figure 4 including all intermediate proof explanation text.

(SET722) $\underset{A,B,C,f,g}{\forall} f :: A \to B \wedge g \circ f :: A \overset{surj.}{\to} C \Rightarrow g :: B \overset{surj.}{\to} C$,

under the assumption:

(Definition (Composition)) $\underset{f,g,x}{\forall} (g \circ f)[x] := g[f[x]]$.

We assume

(1) $f_0 :: A_0 \to B_0 \wedge g_0 \circ f_0 :: A_0 \overset{surj.}{\to} C_0$,

and show

(2) $g_0 :: B_0 \overset{surj.}{\to} C_0$.

In order to show surjectivity of g_0 in (2) we assume

(3) $x1_0 \in C_0$,

and show

(4) $\underset{B1}{\exists} B1 \in B_0 \wedge g_0[B1] = x1_0$.

From (1.1) we can infer

(6) $\underset{A1}{\forall} A1 \in A_0 \Rightarrow f_0[A1] \in B_0$.

From (1.2) we know by definition of "surjectivity"

(7) $\underset{A2}{\forall} A2 \in A_0 \Rightarrow (g_0 \circ f_0)[A2] \in C_0$,

(8) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow \underset{A2}{\exists} A2 \in A_0 \wedge (g_0 \circ f_0)[A2] = x2$.

By (8), we can take an appropriate Skolem function such that

(9) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow A2_0[x2] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x2]] = x2$.

Formula (3), by (9), implies:

$A2_0[x1_0] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (6), implies:

$f_0[A2_0[x1_0]] \in B_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (Definition (Composition)), implies:

(10) $f_0[A2_0[x1_0]] \in B_0 \wedge g_0[f_0[A2_0[x1_0]]] = x1_0$.

Formula (4) is proven because, with $B1 := f_0[A2_0[x1_0]]$, (10) is an instance.

FIGURE 4 A proof generated completely automatically by *Theorema*.

Unification 4

Unification is a fundamental symbolic computation process. Its goal is to identify two given symbolic expressions by means of finding suitable instantiations for certain subexpressions (variables). When the term "identify" is interpreted as syntactic identity, one talks about syntactic unification. If "identify" means equality modulo some given equalities, then is it called equational unification. Hence, unification can be seen as solving equations in abstract algebras, which is used almost everywhere in mathematics and computer science.

Research on unification at RISC has been motivated by its applications in automated reasoning, software engineering, and semistructured data processing. The main subject of study was unification in theories with flexible arity functions and sequence variables, called sequence unification. Such theories are a subject of growing interest as they have been recognized to be useful in various areas, such as XML data modeling with unranked ordered trees and hedges, programming, program transformation, automated reasoning, artificial intelligence, knowledge representation, etc. It is not a surprise that these applications, in some form, require solving equations over terms with flexible arity functions and sequence variables. Hence, sequence unification (and its special forms) play a fundamental role there. Intensive research undertaken at RISC on this subject produced important results that shed light on theoretical and algorithmic aspects of sequence unification, including proving its decidability, developing a solving procedure, identifying important special cases and designing efficient algorithms for them, and finding relations with other unification problems. Some of these results are briefly reviewed below.

General Sequence Unification 4.1

Sequence unification deals with solving systems of equations (unification problems) built over flexible arity function symbols and individual and sequence variables. An instance of such an equation is $f(\overline{x}, x, \overline{y}) = f(f(\overline{x}), x, a, b)$, where f, a, b are function symbols, $\overline{x}, \overline{y}$ are sequence variables, and x is an individual variable. It can be solved by a substitution $\{\overline{x} \mapsto (\,), x \mapsto f, \overline{y} \mapsto (f, a, b)\}$ that maps $\overline{x}$ to the empty sequence, x to the term f (that is a shorthand for $f()$), and $\overline{y}$ to the sequence (f, a, b). Solving systems of such equations can be quite a difficult task: It is not straightforward at all to decide whether a given system has a solution or not. Moreover, some equations may have infinitely many solutions, like, e.g. $f(a, \overline{x}) = f(\overline{x}, a)$ whose solutions are the substitutions $\{\overline{x} \mapsto (\,)\}, \{\overline{x} \mapsto a\}, \{\overline{x} \mapsto (a, a)\}, \ldots$.

When solving unification problems, one is usually interested only in most general solutions from which any solution can be generated. Unification procedures try to compute a (preferably minimal) complete set of such most general unifiers. In the sequence unification case, since for some problems this set can be infinite, any complete unification procedure can only give an enumeration of the set. It can not be used as a decision procedure, in general. Hence, to completely solve sequence unification problems, one needs

1. an algorithm to decide whether a given system of equations is solvable and
2. the procedure that enumerates a minimal complete set of unifiers for solvable systems.

In [Kut07], both of these problems have been addressed. Decidability of sequence unification has been proved by reducing the problem to a combination of word equations and Robinson unification, both with linear constant restrictions. Each of these theories is decidable and the Baader-Schulz combination method [BS96] ensures decidability of the combined theory. Since the reduction from sequence unification to this combined theory is solvability-preserving, the reduction together with the combination method and the decision algorithms for the ingredient theories gives a decision algorithm for sequence unification.

Furthermore, a sequence unification procedure is formulated as a set of rules together with a strategy of their application. If a unification problem is solvable, the procedure nondeterministically selects an equation from the problem and transforms it by all the rules that are applicable. The process iterates for each newly obtained unification problem until a solution is computed or a failure is detected. Since each selected equation can be transformed in finitely many ways, the search tree is finitely branching. However, the tree can still be infinite because some unification problems have infinitely many solutions and the procedure goes on to enumerate them. As it is shown in [Kut07], the procedure generates a minimal and complete set of sequence unifiers and terminates if this set is finite.

As the decision algorithm is quite expensive, it is interesting to identify fragments of sequence unification problems for which the unification procedure terminates without applying the decision algorithm. Several such fragments exist: the linear fragment, where each variable occurs at most once; the linear shallow fragment, which is linear only in sequence variables but restricts them to occur only on level 1 in terms; the fragment where there is no restriction in the number of variable occurrences but sequence variables are allowed to be only the last argument in (sub)terms they occur; sequence matching, where one of the sides of equations is ground (variable-free); the quadratic fragment, where each variable can occur at most twice.

These fragments differ on their unification types that is defined by maximal possible cardinality of minimal complete sets of unifiers of unification problems. Unification problems where sequence variables occur only in the last argument position are of type unitary, which means that if such a prob-

lem is solvable, it has a single most general unifier. It makes this fragment attractive for automated reasoning and, in fact, the Equational Prover of Theorema [Kut03] can deal with it. The quadratic fragment is infinitary (like the general sequence unification itself), which means that there are some solvable problems with an infinite minimal complete set of unifiers. The equation $f(a, \overline{x}) = f(\overline{x}, a)$ above is an example of such a quadratic problem. However, a nice thing is that, for quadratic problems, one can represent these infinite sets by finite means, in particular, as regular expressions over substitutions. The quadratic fragment has found an application in collaborative schema development in the joint work of T. Kutsia (RISC), M. Florido and J. Coelho (both from Portugal) [CFK07]. All the other mentioned fragments are finitary: For them, solvable unification problems may have at most finitely many most general unifiers.

These fragments have interesting properties and applications. Two of them have already been mentioned above. Among others, the sequence matching capabilities of the Mathematica system [Wol03] should be noted, which makes the programming language of Mathematica very flexible.

It should be noted that all the results on sequence unification in [Kut07], in fact, have been formulated in a more general setting: besides function symbols and individual and sequence variables, the problems may contain so called sequence functions. A sequence function abbreviates a finite sequence of functions all having the same argument lists. Semantically, they can be interpreted as multi-valued functions. Bringing sequence functions into the language allows Skolemization over sequence variables. For instance, $\forall\overline{x}\exists\overline{y}\ p(\overline{x}, \overline{y}))$ after Skolemization introduces a sequence function symbol $\overline{g}$: $\forall\overline{x}\ p(\overline{x}, \overline{g}(\overline{x}))$. From the unification point of view, a sequence function can be split between sequence variables. The corresponding rules are part of the unification procedure described in [Kut07].

Flat Matching 4.2

Sequence matching problems, as already mentioned, are those that have a ground side in the equations. An instance of such an equation is $f(x, \overline{y}) = f(a, b, c)$ which has a single solution (matcher) $\{x \mapsto a, \overline{y} \mapsto (b, c)\}$. But what happens if f satisfies the equality $f(\overline{x}, f(\overline{y}), \overline{z}) = f(\overline{x}, \overline{y}, \overline{z})$, i.e. if one can flatten out all nested occurrences of f? It turns out that in such a case the minimal complete set of matchers becomes infinite. The substitutions like $\{x \mapsto f(), \overline{y} \mapsto (f(), a, b, c)\}$, $\{x \mapsto f(), \overline{y} \mapsto (a, f(), b, c)\}$, $\{x \mapsto f(), \overline{y} \mapsto (f(), a, f(), b, c)\}$ and similar others become solutions modulo flatness of f. It is quite unusual for matching problems to have an infinite minimal complete set of solutions. It triggered our interest to matching in flat theories, to study

theoretical properties of flat matching, to design a complete procedure to solve flat matching problems, and to investigate terminating restrictions.

But this was only one side of the problem. On the other side, a flat theory is not a theory that is "cooked artificially" to demonstrate that matching problems can be arbitrarily complex. It has a practical application: Flat symbols appear in the programming language of the Mathematica system, by assigning to certain symbols the attribute `Flat`. This property affects both evaluation and pattern matching in Mathematica. Obviously, a practically useful method that solves flat matching equations should be terminating and, therefore, incomplete (unless it provides a finite description of the infinite complete set of flat matchers). Understanding proper semantics of programming constructs is very important to program correctly. Hence, the questions arise: What is the semantics of Mathematica's incomplete flat matching algorithm? What are the rules behind it, how it works? How is the algorithm related to theoretically complete, infinitary flat matching? These questions have not been formally answered before.

[Kut08] addresses both theoretical and practical sides of the problem. From the theoretical side, it gives a procedure to solve a system of flat matching equations and proves its soundness, completeness, and minimality. The minimal complete set of matchers for such a system can be infinite. The procedure enumerates this set and stops if it is finite. Besides, a class of problems on which the procedure stops is described. From the practical point of view, it gives a set of rules to simulate behavior of the flat matching algorithm implemented in the Mathematica system.

Differences between various flat matching procedures can be demonstrated on simple examples. For instance, given a problem $\{f(\overline{x}) = f(a)\}$ where f is flat, the minimal complete flat matching procedure enumerates its infinite minimal complete set of matchers $\{\overline{x} \mapsto a\}, \{\overline{x} \mapsto f(a)\}, \{\overline{x} \mapsto (f(), a)\}, \{\overline{x} \mapsto (a, f())\}, \{\overline{x} \mapsto (f(), f(), a)\}, \ldots$. Restricting the rules in the procedure so that $f()$ is not generated in such cases, one obtains a terminating incomplete algorithm that returns two matchers $\{\overline{x} \mapsto a\}, \{\overline{x} \mapsto f(a)\}$. In order to simulate Mathematica's flat matching, further restrictions should be imposed on the rules to obtain the only matcher $\{\overline{x} \mapsto a\}$. It should be noted that Mathematica's behavior depends whether one has a sequence variable or an individual variable under the flat function symbol. Also, Mathematica treats in a special way function variables (those that can be instantiated with function symbols). [Kut08] analyzes all those cases and gives a formal description of the corresponding rules.

Context Sequence Matching 4.3

Flat matching (and, in general, matching modulo equations with sequence variables) is one generalization of syntactic sequence matching. Another generalization comes from bringing higher-order variables in the terms. T. Kutsia (RISC) in collaboration with M. Marin (Japan) studied extension of sequence matching with function and context variables [KM05, Kut06]. Function variables have already been mentioned above. Context variables are second-order variables that can be instantiated with a context—a term with a single occurrence of a distinguished constant $\bullet$ (called the hole) in it. A context can be applied to a term by replacing the hole with that term. An example of context sequence matching equation is $\overline{X}(f(\overline{x})) = g(f(a,b), h(f(a), f)$, where $\overline{X}$ is a context variable and $\overline{x}$ is a sequence variable. Its minimal complete set of matchers consists of three elements: $\{\overline{X} \mapsto g(\bullet, h(f(a), f)),\ \overline{x} \mapsto (a,b)\}$, $\{\overline{X} \mapsto g(f(a,b), h(\bullet, f)),\ \overline{x} \mapsto a\}$, and $\{\overline{X} \mapsto g(f(a,b), h(f(a), \bullet)),\ \overline{x} \mapsto ()\}$.

Context sequence matching is a flexible mechanism to extract subterms from a given ground term via traversing it both in breadth and in depth. Function variables allow to descend in depth in one step, while with context variables subterms can be searched in arbitrary depth. Dually, individual variables and sequence variables allow moves in breadth: individual variables in one step and sequence variable in arbitrary number of steps. This duality makes context sequence matching an attractive technique for expressing subterm retrieval queries in a compact and transparent way.

Context and sequence variables occurring in matching problems can be constrained by membership atoms. Possible instantiations of context variables are constrained to belong to a regular tree language, whereas the ones for sequence variables should be elements of regular hedge languages. This extension is the main computational mechanism for the experimental rule-based programming package ρLog [MK06].

Relations between Context and Sequence Unification 4.4

Context unification [Com91, SS94] aims at solving equations for terms built over fixed arity function symbols and first-order and context variables. It is one of the most difficult problems in unification theory: Its decidability is an open problem already for more than 15 years. There have been various decidable fragments (obtained by restricting the form of the input equations) and variants (obtained by restricting the form of possible solutions) identified; see, e.g. [Com98, Lev96, SSS02, LNV05] and for more comprehensive overview, [Vil04]. Both sequence unification and context unification generalize the well-known word unification problem [Mak77]. One of them is decidable,

while decidability of the other one is an open problem. Hence, a natural question arises: How are these two generalizations of the same problem related with each other?

T. Kutsia (RISC), J. Levy and M. Villaret (both from Spain) gave a complete answer to this problem in [KLV07]. First, they defined a mapping (called curryfication) from sequence unification to a fragment of context unification such that if the original sequence unification problem is solvable, then the curried context unification problem is also solvable. However, this transformation does not preserve solvability in the other direction. To deal with this problem, possible solutions of curried context unification problems have been restricted to have a certain shape, called left-hole context, which can be characterized by the property of having holes in the leftmost leaf in their tree representation, like, for instance, in the context $@(@(\bullet, a), b)$. (In curried problems @ is the only binary function symbol and all the other function symbols are constants, but it is not a restriction for solvability, at it was shown in [LV02].) This restriction guarantees solvability preservation between sequence unification and the corresponding fragment of context unification. Next, the left-hole restriction has been extended from the fragment to the whole problem, obtaining a variant, called left-hole context unification (LHCU). To prove solvability of LHCU, another transformation has been defined that transforms LHCU into word equations with regular constraints. The transformation is solvability-preserving and word unification with regular constraints is decidable, which implies decidability of LHCU. Finally, transforming LHCU with inverse curryfication, a decidable extension of sequence unification has been obtained. This transformation also made it possible to transfer some of the known complexity results for context matching to extended sequence matching.

Hence, this work can be summarized as follows: A new decidable variant of context unification has been discovered; A decidable extension of sequence unification has been found and a complete unification procedure has been developed; A new proof of decidability of sequence unification has been given; Complexity results for (some fragments of) extended sequence matching have been formulated.

5 Program Verification

The activities related to program verification in the *Theorema* group refer to various programming styles and to various verification techniques. The *Theorema* system allows to describe algorithms directly in predicate logic, which is sometimes called "pattern based programming". Using some abbreviating constructs (as e. g. `if-then-else`), in *Theorema* one can also use the functional programming style. In both cases the verification benefits from the fact that the properties of the programs are expressed in the same logical

language, thus a possibly error prone translation is not necessary. Furthermore, in order to experiment with alternative techniques, *Theorema* provides additionally a simple language for imperative programming.

In this section we focus on the verification of functional programs, however the research on verification of imperative programs is also strongly pursued by our group. For instance, the work on *loop invariants* lead to a complex method which uses algebraic and combinatorial techniques for the automatic generation of polynomial invariants of `while` loops [KPJ05, Kov07]. A very novel and interesting aspect of this method is the nontrivial interplay between logical techniques on one hand, and algebraic techniques on the other hand, which demonstrates the high value of the approach of combining automated reasoning with computer algebra into the field of symbolic computation. Moreover, the recent research on *symbolic execution* [EJ08] introduces a novel approach to the generation of verification conditions exclusively in the theory of the domain of the objects handled by the program—including the termination condition.

Some Principles of Program Verification 5.1

Before a more detailed presentation of our research, we summarize shortly some main principles of program verification. Note that we focus here on the techniques which are based on automated theorem proving, and not, for instance, on model checking techniques.

Program specification (or formal specification of a program) is the definition of what a program is expected to do. Normally, it does not describe, and it should not, how the program is implemented. The specification is usually provided by logical formulae describing a relationship between input and output parameters. We consider specifications which are pairs, containing a precondition (input condition) and a postcondition (output condition).

Formal verification consists in proving mathematically the correctness of a program with respect to a certain formal specification. Software testing, in contrast to verification, cannot prove that a system does not contain any defects or that it has a certain property.

The problem of verifying programs is usually split into two subproblems: *generate* verification conditions which are sufficient for the program to be correct and *prove* the verification conditions, within the theory of the domain for which the program is defined. A survey of the techniques based on this principle, but also of other techniques can be found e. g. in [LS87] and in [Hoa03].

A Verification Condition Generator (VCG) is a device—normally implemented by a program—which takes a program, actually its source code, and the specification, and produces verification conditions. These verification con-

ditions do not contain any part of the program text, and are expressed in a different language, namely they are logical formulae.

Normally, these conditions are given to an automatic or semi-automatic theorem prover. If all of them hold, then the program is correct with respect to its specification. The latter statement we call *Soundness* of the VCG, namely:

Given a program F and a specification I_F (input condition), and O_F (output condition), if the verification conditions generated by the VCG hold, then the program F is correct with respect to the specification $\langle I_F, O_F \rangle$.

Completing the notion of *Soundness* of a VCG, we introduce its dual—*Completeness* [KPJ06]:

Given a program F and a specification I_F (input condition), and O_F (output condition), if the program F is correct with respect to the specification $\langle I_F, O_F \rangle$, then the verification conditions generated by the VCG hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong". Indeed, most of the literature on program verification presents methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

A distinction is to be made between total correctness, which additionally requires that the program terminates, and partial correctness, which simply requires that if an answer is returned (that is, the program terminates) it will be correct. Termination is in general more difficult. On one hand, it is theoretically proven that termination is not decidable in general (however this does not mean that we cannot prove termination of specific programs). On the other hand, the statement "program P terminates" is difficult or impossible to express in the theory of the domain of the program, but has to be introduced additionally. Adding a suitable theory of computation will increase significantly the formalization and the proving effort. Our approach to this problem is to decompose the total correctness into many simpler formulae (the verification conditions), and to reduce termination of the original program to the termination of a simplified version of it, as shown in the sequel.

5.2 Verification of Functional Programs

In the *Theorema* system we see functional programs as abbreviations of logical formulae (for instance, an `if-then-else` clause is an abbreviation of two implications). Therefore, the programming language is practically identical

to the logical language which is used for the verification conditions. This has the advantage that we do not need to translate the predicate symbols and the function symbols occurring in the program: they are already present in the logical language.

Our work consists in developing the theoretical basis and in implementing an experimental prototype environment for defining and verifying recursive functional programs. In contrast to classical books on program verification [Hoa69], [BL81], [LS87] which expose methods for verifying correct programs, we also emphasize the detection of incorrect programs. The user may easily interact with the system in order to correct the program definition or the specification.

We first perform a check whether the program under consideration is *coherent* with respect to the specification of its components, that is, each function is applied to arguments satisfying its input condition. (This principle is also known as *programming by contract*.)

The program correctness is then transformed into a set of first-order predicate logic formulae by a Verification Condition Generator (VCG)—a device, which takes the program (its source code) and the specification (precondition and postcondition) and produces several verification conditions, which themselves, do not refer to any theoretical model for program semantics or program execution, but only to the theory of the domain used in the program.

For coherent programs we are able to define a necessary and sufficient set of verification conditions, thus our condition generator is not only *sound*, but also *complete*. This distinctive feature of our method is very useful in practice for program debugging.

Since coherence is enforced, verification can be performed independently on different programs, thus one avoids the costly process of *interprocedural analysis*, which is sometimes used in model checking. Moreover, the correctness of the whole system is preserved even when the implementation of a function is changed, as long as it still satisfies the specification.

In order to illustrate our approach, we consider powering function P, using the *binary powering* algorithm:

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

This program is in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x,n] = x^n. \tag{5}$$

The (automatically generated) conditions for *coherence* are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow \mathbb{T}) \tag{6}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow \mathrm{Even}[n]) \tag{7}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow \mathrm{Odd}[n]) \tag{8}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{9}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow \mathbb{T}) \tag{10}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{11}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{12}$$

One sees that the formulae (6), (9) and (10) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (7), (8), (11) and (12) are easy consequences of the elementary theory of reals and naturals. For the further check of *correctness* the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 1 = x^n) \tag{13}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2}\ \Rightarrow m = x^n) \tag{14}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n) \tag{15}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = \mathbb{T}, \tag{16}$$

where

$$\begin{aligned} P'[x, n] = \ & \textbf{If } n = 0 \textbf{ then } \mathbb{T} \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P'[x * x, n/2] \\ & \textbf{else } P'[x * x, (n-1)/2]. \end{aligned}$$

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, for this buggy version of P we may see that all the respective verification conditions remain the same except one, namely, (13) is now:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{17}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debugging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (14) is now:

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{18}$$

which itself reduces to:

$$m = x^{n/2}\ \Rightarrow m = x^n$$

From here, we see that the "Even" branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2}\ \Rightarrow m = x^n$$

which finally leads to the correct version of P.

6 Computer-Assisted Interactive Program Reasoning

As demonstrated in the other sections of this chapter, much progress has been made in automated reasoning and its application to the verification of computer programs and systems. In practice however, for programs of a certain complexity, fully automatic verifications are not feasible; much more success is achieved by the use of *interactive proving assistants* which allow the

user to guide the software towards a semi-automatic construction of a proof by iteratively applying predefined proof decomposition strategies in alternation with critical steps that rely on the user's own creativity. The goal is to reach proof situations that can be automatically closed by *SMT (satisfiability modulo theories) solvers* [SMT06] which decide the truth of unquantified formulas over certain combinations of ground theories (uninterpreted function symbols, linear integer arithmetic, and others). In a modern computer science education, it is important to train students in the use of such systems which can help in formal specifying programs and reasoning about their properties.

The RISC ProofNavigator

While a variety of tools for supporting reasoning are around, many of them are difficult to learn and/or inconvenient to use, which makes them less suitable for classroom scenarios [Fei05]. This was also Schreiner's experience when he evaluated from 2004 to 2005 a couple of prominent proving assistants by a number of use cases derived from the area of program verification. While he achieved quite good results with PVS [ORS92], he generally encountered various problems and nuisances, especially with the navigation within proofs, the presentation of proof states, the treatment of arithmetic, and the general interaction of the user with the systems; he frequently found that the elaboration of proofs was more difficult than should be necessary.

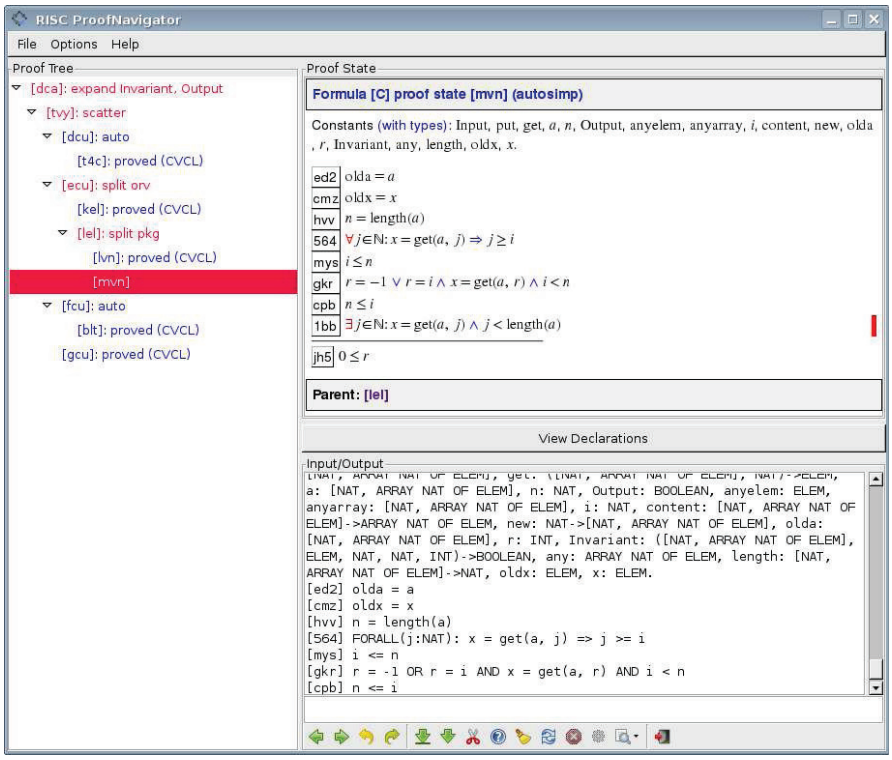
Based on these investigations, Schreiner developed the *RISC ProofNavigator* [RIS06, Sch08b], a proving assistant which is intended for educational scenarios but has been also applied to verifications that are already difficult to handle with other assistants. The software currently applies the *Cooperating Validity Checker Lite (CVCL)* [BB04] as the underlying SMT solver. Its user interface (depicted in Figure 5) was designed to meet various goals:

Maximize Survey: The user should easily keep a general view on proofs with many states; she should also easily keep control on proof states with large numbers of potentially large formulas. Every proof state is automatically simplified before it is presented to the user.

Minimize Options: The number of commands is kept as small as possible in order to minimize confusion and simplify the learning process (in total there are about thirty commands, of which only twenty are actually proving commands; typically, less than ten commands need to be used).

Minimize Efforts: The most important commands can be triggered by buttons or by menu entries attached to formula labels. The keyboard only needs to be used in order to enter terms for specific instantiations of universal assumptions or existential goals.

The proof of a verification condition is displayed in the form of a tree structure such as the following proof of a condition arising from the verification of the linear search algorithm [Sch06]:

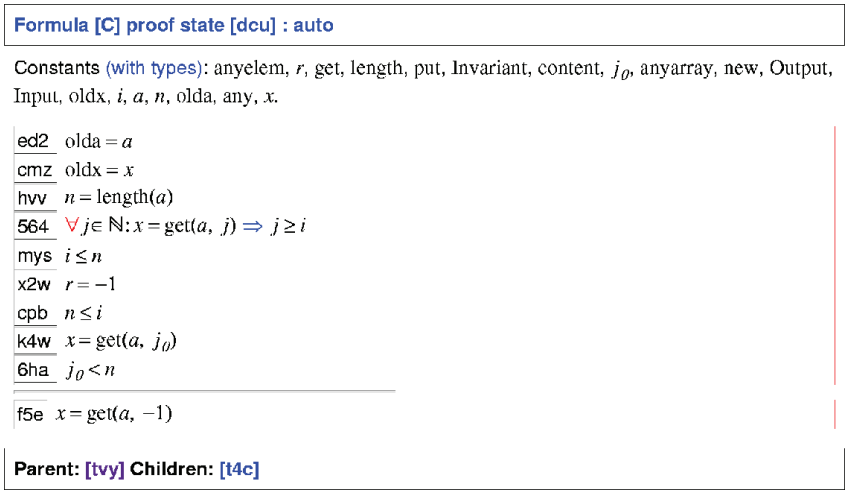


The RISC ProofNavigator in action. FIGURE 5

```
[dca]: expand Invariant, Output in zfg
  [tvy]: scatter
    [dcu]: auto
      [t4c]: proved (CVCL)
    [ecu]: split pkg
      [kel]: proved (CVCL)
      [lel]: scatter
        [lvn]: auto
          [lap]: proved (CVCL)
    [fcu]: auto
      [blt]: proved (CVCL)
    [gcu]: proved (CVCL)
```

Here the user expands predicate definitions (command `expand`), performs automatic proof decomposition (command `scatter`), splits a proof situation based on a disjunctive assumption (command `split`), performs automatic instantiation of a quantified formula (command `auto`), and thus reaches proof situations that can be automatically closed by CVCL. Each proof situation is displayed as a list of assumptions from which a particular goal is to be

proved (the formula labels represent active menus from which appropriate proof commands can be selected):



The software is used since 2007 in regular courses offered to students of computer science and mathematics at the Johannes Kepler University Linz and at the Upper Austria University of Applied Sciences Campus Hagenberg; it is freely available as open source and shipped with a couple of examples:

1. Induction proofs,
2. Quantifier proofs,
3. Proofs based on axiomatization of arrays,
4. Proofs based on constructive definition of arrays,
5. Verification of linear search,
6. Verification of binary search,
7. Verification of a concurrent system of one server and 2 clients,
8. Verification of a concurrent system of one server and N clients.

The last two proofs consist of some hundreds of situations (most of which are closed automatically, the user has to apply about two dozens commands only) and were hard/impossible to manage with some other assistants.

The RISC ProgramExplorer

The RISC ProofNavigator is envisioned as a component of a future environment for formal program analysis, the RISC ProgramExplorer, which is currently under development. Unlike program verification environments (such as KeY [BHS07]) which primarily aim at the automation of the verification process, the goal of this environment is to *exhibit* the logical interpretation of imperative programs and *clarify* the relationship between reasoning tasks one one side and program specifications/implementations on the other side, and

thus *assist* the user in analyzing a program and establishing its properties. The core features of this environment will be

1. a translation of programs to logical formulas that exhibit the semantic essence of programs as relations on pairs of (input/output) states [Sch08a], e.g. the program

   ```
   { var i; i = x+1; x = 2*i; }
   ```

   becomes the formula

   $$\exists i, i' : i' = x + 1 \wedge x' = 2 \cdot i'$$

   which can be simplified to $x' = 2x + 2$;
2. the association of verification conditions to specific program positions (respectively execution paths in the program) such that failures in verifications can be more easily related to programming errors.

The environment shall support the following tasks:

- Translating programs to formulas which can be subsequently simplified to exhibit the program's semantic essence;
- Validating specifications by verifying that they satisfy given input/output examples, that they are not trivial, and they are implementable,
- Verifying that the program does not violate the preconditions specified for program functions and atomic operations,
- Verifying that the program ensures the specified postconditions,
- Verifying the correctness of (loop/system) invariants,
- Verifying termination of loops and recursive functions,
- Verifying the correctness of abstract datatype representations.

Particular emphasis is given to a graphical user interface that adequately exhibits the duality between the operational and the logical interpretation of programs and the relationship of verification conditions to properties of particular program parts. A first skeleton prototype of this environment will become available in 2009.

Acknowledgements

The research described in this chapter has been performed in the frame of the following research projects at RISC:

- Austrian Science Foundation (FWF) under Project SFB F1302 (Theorema).
- European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIEnce—Symbolic Computation Infrastructure for Europe (Contract No. 026133).

- European Commission Framework 5 Proj Nr. HPRN-CT-2000-00102 (Calculemus).
- INTAS project 05-1000008-8144 "Practical Formal Verification Using Automated Reasoning and Model Checking".
- Upper Austrian Government project "Technologietransferaktivitäten".
- Project "Institute e-Austria Timisoara".

References

[AH77] K. Appel and W. Haken. Solution of the four color map problem. *Scientific American*, 237:108–121, October 1977.

[BB04] C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[BCJ+06] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[BDF+04] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. F. Nilsson. Synthesis of Programs in Computational Logic. In M. Bruynooghe and K. K. Lau, editors, *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 30–65. Springer, 2004.

[BDJ+00] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema Project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997)*, pages 384–391. ACM Press, 1997.

[BL81] B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition, 1981.

[BS96] F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation*, 21(2):211–244, 1996.

[Buc65] B. Buchberger. *An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal*. PhD thesis, University Innsbruck, Mathematical Institute, 1965. German, English translation in: J. of Symbolic Computation, Special Issue on Logic, Mathematics, and Computer Science: Interactions. Volume 41, Number 3–4, Pages 475–511, 2006.

[Buc96a] B. Buchberger. Mathematica as a Rewrite Language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Functional and Logic Programming (Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming,*

November 1-4, 1996, Shonan Village Center), pages 1–13. Copyright: World Scientific, Singapore - New Jersey - London - Hong Kong, 1996.

[Buc96b] B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In F. Bader and K.U. Schulz, editors, *Frontiers of Combining Systems, Proceedings of FROCOS 1996 (1st International Workshop on Frontiers of Combining Systems), March 26-28, 1996, Munich*, volume Vol.3 of *Applied Logic Series*, pages 193–220. Kluwer Academic Publisher, Dordrecht - Boston - London, The Netherlands, 1996.

[Buc96c] B. Buchberger. Using Mathematica for Doing Simple Mathematical Proofs. In *Proceedings of the 4th Mathematica Users' Conference, Tokyo, November 2, 1996.*, pages 80–96. Copyright: Wolfram Media Publishing, 1996.

[Buc97] B. Buchberger. Mathematica: Doing Mathematics by Computer? In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 2–20. Springer Vienna, 1997. RISC Book Series on Symbolic Computation.

[Buc99] Bruno Buchberger. Theory Exploration Versus Theorem Proving. Technical Report 99-46, RISC Report Series, University of Linz, Austria, July 1999. Also available as SFB Report No. 99-38, Johannes Kepler University Linz, Spezialforschungsbereich F013, December 1999.

[Buc01] B. Buchberger. The PCS Prover in Theorema. In R. Moreno-Diaz, B. Buchberger, and J.L. Freire, editors, *Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory – Formal Methods and Tools for Computer Science)*, Lecture Notes in Computer Science 2178, pages 469–478. Las Palmas de Gran Canaria, Copyright: Springer - Verlag Berlin, 19-23 February 2001.

[Buc03] B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2003, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 2–26, Timisoara, Romania, 1-4 October 2003. Copyright: Mirton Publisher.

[Buc04] B. Buchberger. Towards the Automated Synthesis of a Gröbner Bases Algorithm. *RACSAM (Rev. Acad. Cienc., Spanish Royal Academy of Science)*, 98(1):65–75, 2004.

[Buc06] B. Buchberger. Mathematical Theory Exploration, August 17-20 2006. Invited talk at IJCAR, Seattle, USA.

[CFK07] J. Coelho, M. Florido, and T. Kutsia. Sequence disunification and its application in collaborative schema construction. In M. Weske, M.-S. Hacid, and C. Godart, editors, *Web Information Systems – WISE 2007 Workshops*, volume 4832 of *LNCS*, pages 91–102. Springer, 2007.

[Col75] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Second GI Conference on Authomata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.

[Com91] H. Comon. Completion of rewrite systems with membership constraints. Rapport de Recherche 699, L.R.I., Université de Paris-Sud, 1991.

[Com98] H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *Journal of Symbolic Computation*, 25(4):421–453, 1998.

[Cra08] A. Craciun. *Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory*. PhD thesis, RISC, Johannes Kepler University Linz, Austria, April 2008.

[EJ08] M. Erascu and T. Jebelean. Practical Program Verification by Forward Symbolic Execution: Correctness and Examples. In B. Buchberger, T. Ida, and T. Kutsia, editors, *Austrian-Japan Workshop on Symbolic Computation in Software Science*, pages 47–56, 2008.

[Fei05] Ingo Feinerer. Formal Program Verification: A Comparison of Selected Tools and Their Theoretical Foundations. Master's thesis, Theory and Logic Group,

Institute of Computer Languages, Vienna University of Technology, Vienna, Austria, January 2005.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of ACM*, 50:63–69, 2003.

[Jeb01] T. Jebelean. Natural proofs in elementary analysis by S-Decomposition. Technical Report 01-33, RISC, Johannes Kepler University, Linz, Austria, 2001.

[KJ00] B. Konev and T. Jebelean. Using meta-variables for natural deduction in theorema. In M. Kerber and M. Kohlhase, editors, *Proceedings of the CALCULEMUS 2000 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, pages 160–175, St. Andrews, Scotland, August 6-7 2000.

[KLV07] T. Kutsia, J. Levy, and M. Villaret. Sequence unification through currying. In Franz Baader, editor, *Proc. of the 18th Int. Conference on Rewriting Techniques and Applications, RTA'07*, volume 4533 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2007.

[KM05] T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. Proceedings of the 12th International Conference, LPAR'05*, volume 3835 of *LNAI*, pages 215–229. Springer, 2005.

[Kov07] L. Kovacs. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema*. PhD thesis, RISC, Johannes Kepler University Linz, Austria, October 2007. RISC Technical Report No. 07-16.

[KPJ05] L. Kovacs, N. Popov, and T. Jebelean. Verification Environment in Theorema. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(2):27–34, 2005.

[KPJ06] L. Kovacs, N. Popov, and T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In T. Margaria and B. Steffen, editors, *Proceedings ISOLA 2006*, Paphos, Cyprus, November 2006. To appear.

[Kut03] T. Kutsia. Equational prover of Theorema. In R. Nieuwenhuis, editor, *Proc. of the 14th Int. Conference on Rewriting Techniques and Applications, RTA'03*, volume 2706 of *LNCS*, pages 367–379. Springer, 2003.

[Kut06] T. Kutsia. Context sequence matching for XML. *Electronic Notes on Theoretical Computer Science*, 157(2):47–65, 2006.

[Kut07] T. Kutsia. Solving equations with sequence variables and sequence functions. *Journal of Symbolic Computation*, 42(3):352–388, 2007.

[Kut08] T. Kutsia. Flat matching. *Journal of Symbolic Computation*, 43(12):858–873, 2008.

[Lev96] J. Levy. Linear second-order unification. In Harald Ganzinger, editor, *Proc. of the 7th Int. Conference Conference on Rewriting Techniques and Applications, RTA'96*, volume 1103 of *LNCS*, pages 332–346. Springer, 1996.

[LNV05] J. Levy, J. Niehren, and M. Villaret. Well-nested context unification. In R. Nieuwenhuis, editor, *Proc. of the 20th Int. Conference on Automated Deduction, CADE-20*, volume 3632 of *LNAI*, pages 149–163. Springer, 2005.

[LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Teubner, second edition, 1987.

[LV02] J. Levy and M. Villaret. Currying second-order unification problems. In S. Tison, editor, *Proc. of the 13th Int. Conference on Rewriting Techniques and Applications, RTA'02*, volume 2378 of *LNCS*, pages 326–339, Copenhagen, Denmark, 2002. Springer.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977.

[McC97] W. McCune. Solution of the robbins problem. *Journal of Automatic Reasoning*, 19:263–276, 1997.

[MK06] M. Marin and T. Kutsia. Foundations of the rule-based system RhoLog. *Journal of Applied Non-Classical Logics*, 16(1–2):151–168, 2006.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 14–18, 1992. Springer.

[PB02] F. Piroi and B. Buchberger. Focus Windows: A New Technique for Proof Presentation. In H. Kredel and W. Seiler, editors, *Proceedings of the 8th Rhine Workshop on Computer Algebra, Mannheim, Germany*, pages 297–313, 2002.

[PK05] Florina Piroi and Temur Kutsia. The Theorema Environment for Interactive Proof Development, December 3 2005. Contributed talk at 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'05.

[RIS06] The RISC ProofNavigator, 2006. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator.

[Rob02] Judit Robu. Geometry Theorem Proving in the Frame of the Theorema Project. Technical Report 02-23, RISC Report Series, University of Linz, Austria, September 2002. PhD Thesis.

[Sch06] W. Schreiner. The RISC ProofNavigator — Tutorial and Manual. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, July 2006.

[Sch08a] W. Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008.

[Sch08b] W. Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, April 2008. DOI 10.1007/s00165-008-0069-4.

[SMT06] SMT-LIB — The Satisfiability Modulo Theories Library, 2006. University of Iowa, Iowa City, IA, http://combination.cs.uiowa.edu/smtlib.

[SS94] M. Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/24, Johann-Wolfgang-Goethe-Universität, Frankfurt, Germany, 1994.

[SSS02] M. Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *Journal of Symbolic Computation*, 33(1):77–122, 2002.

[Vil04] M. Villaret. *On Some Variants of Second-Order Unification*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 2004.

[VJB08] R. Vajda, T. Jebelean, and B. Buchberger. Combining Logical and Algebraic Techniques for Natural Style Proving in Elementary Analysis. *Mathematics and Computers in Simulation*, 2008.

[WBR06] W. Windsteiger, B. Buchberger, and M. Rosenkranz. Theorema. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *LNAI*, pages 96–107. Springer Berlin Heidelberg New York, 2006.

[Win01] W. Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*. PhD thesis, RISC Institute, May 2001.

[Win06] W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. *JSC*, 41(3-4):435–470, 2006.

[Wol03] S. Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.