JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory

Dissertation

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

Angefertigt am *Institut für Symbolisches Rechnen*

Betreuung:

Erster Begutachter: o.Univ.-Prof.Dr. Dr.h.c.mult. Bruno Buchberger
Zweiter Begutachter: o.Univ.-Prof.Dr. Dr.h.c. Günter Pilz

Eingereicht von:

Vasile Adrian Crăciun, M.Sc.

Linz, März 2008

Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory

Doctoral Thesis

Vasile Adrian Crăciun, M.Sc.

Advised by

o.Univ.Prof.Dr. Dr.h.c.mult. Bruno Buchberger
o.Univ.Prof.Dr. Dr.h.c. Günter Pilz

Research Institute for Symbolic Computation
Johannes Kepler University Linz

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörthlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Vasile Adrian Crăciun
Linz, März 2007

Zusammenfassung

Diese Dissertation befasst sich mit der Implementierung der Synthesemethode "Lazy Thinking" und ihrer Anwendung, Algorithmen in der Theorie der Gröbner-Basen im Rahmen des mathematischen Systems Theorema zu generieren.

Die Methode "Lazy Thinking" wurde von Bruno Buchberger, dem Erstbegutachter dieser Dissertation, eingeführt und ist Teil seines Modells der systematischen Exploration mathematischer Theorien, die auf Formelschemata basiert.

Im Wesentlichen ist "Lazy Thinking" eine deduktive, schemabasierte Synthesemethode. Zuerst wird versucht, die Korrektheit eines Algorithmus (d.h. die Erfüllung seiner Spezifikation) zu beweisen, indem ein Algorithmenschema und das gesamte Wissen über die Spezifikation verwendet werden. Dabei sind sämtliche Definitionen und Eigenschaften der in den Spezifikationen involvierten Begriffe bekannt. Das Algorithmusschema ist hierbei eine Definition des gesuchten Algorithmus, wobei die auftretenden Unteralgorithmen noch nicht näher spezifiziert sind. Deshalb wird der Beweisversuch höchstwahrscheinlich scheitern. Nach der Analyse des gescheiterten Beweisversuchs werden Vermutungen aufgestellt und der Wissenbasis hinzugefügt, so dass schließlich der Beweis zum Erfolg geführt werden kann. Diese Vermutungen stellen sich dabei als Spezifikationen für die unbekannten Unteralgorithmen heraus. Algorithmen, die diese generierten Spezifikationen erfüllen, können dann entweder aus einer Wissensdatenbank geholt oder rekursiv wieder mit "Lazy Thinking" in weiteren Explorationszyklen generiert werden.

In dieser Arbeit stellen wir die Methode von "Lazy Thinking" und deren Implementierung in *Theorema* vor:

- der *Kaskadenmechanismus*, der die "Lazy"-Explorationszyklen (Initiierung des Beweises, Analyse des Beweismisserfolgs, Aufstellung der Vermutung) implementiert,
- die *Misserfolgsanalyse* und

- die *Aufstellung der Vermutungen.*

Anschließend beschreiben wir die Verwendung des Kaskadenmechanismus in Interaktion mit den vorhandenen Beweisern von Theorema.

Das Gröbnerbasenproblem stellt einen höchst nicht-trivialen Tauglichkeitstest für die automatische Algorithmensynthese dar.

Basierend auf Bruno Buchbergers vorgeschlagenem Leitfaden wird vorgestellt, wie die Implementierung von "Lazy Thinking" in diesem Fallbeispiel verwendet werden kann. Wir starten dabei mit dem Algorithmusschema, das die Vervollständigung der kritischen Polynompaare realisiert. Unter Verwendung dieses Schemas wird gezeigt, dass unsere vorgestellte Implementierung erfolgreich angewendet werden kann, um die Idee der S-Polynome (wie von Bruno Buchberger ursprünglich eingeführt) effektiv neu zu erfinden und einen Korrektheitsbeweis erfolgreich abzuschließen.

Abstract

This thesis is concerned with the implementation of the "lazy thinking" synthesis method in the frame of the mathematical assistant system *Theorema* and its application to the synthesis of algorithms in the theory of Gröbner bases.

The "lazy thinking" method, proposed by Bruno Buchberger, the first advisor of this thesis, is part of his model of systematic theory exploration based on knowledge schemes.

Essentially, lazy thinking is a deductive, scheme-based synthesis method: a proof of the correctness of an algorithm (i.e. its specification) using an algorithm scheme and using complete knowledge about the specification is attempted. All definitions and properties of concepts involved in the specifications are known. The algorithm scheme is a definition of the desired algorithm in terms of unknown subalgorithms. The proof is likely to fail, as no information about the unknown subalgorithms is available. Following an analysis of the failing proof, conjectures are generated and added to the knowledge, such that the failure can be overcome. These conjectures turn out to be specifications for the unknown subalgorithms. Algorithms that satisfy the generated specifications can then either be retrieved from the knowledge base, or synthesized by lazy thinking in subsequent rounds of exploration.

We describe the method of lazy thinking and then describe its implementation in *Theorema*:

- *the cascade mechanism*, that implements the lazy exploration cycles (initiate proof, proof failure analysis, conjecture generation),
- *failure analysis*, and
- *conjecture generation*.

We then describe the usage of the cascade mechanism in conjunction with the reasoners available in the system.

The problem of Gröbner bases represents a nontrivial benchmark for program synthesis. We present how, following an outline proposed by Bruno Buchberger, the lazy thinking implementation can be used in this case study. Starting from a critical-pair completion algorithm scheme, we show how our implementation can be applied successfully to effectively (re)invent the idea of S-polynomials (as outlined by Bruno Buchberger), and complete a proof of correctness.

Acknowledgements

I would first like to thank Bruno Buchberger, my advisor: For starting RISC and devoting so much energy to the institute, making it the great working place it was for me, for starting and leading *Theorema*, for giving me this interesting topic, for his guidance and patience, for the time and effort he put in starting and supporting the e-Austria Institute of Timişoara, where I wrote part of this thesis. Meeting Bruno had a big influence on me. Not only did I learn logic, mathematics, how to do research (and I hope and try to get better at all of these), but what I learned has helped clarify the way I think, the way I understand myself and my place in the "world outside".

My gratitude to Prof. Günter Pilz, for kindly agreeing to be the second advisor to this thesis.

To my professors at the West University of Timişoara, Dana Petcu, Ştefan Măruşter, Viorel Negru, many thanks. Not only did they first show me that research is something one (student) may do (which was not at all clear to me at the time), but also pointed me towards RISC and helped with my application. Also, thanks for having me back in Timişoara, at the University and the e-Austria Institute.

To the (past and present) members of *Theorema* (I got to work with), thank you for all the help during my stay in RISC, interesting discutions and your friendship: Gabor Kusper, Mircea Marin, Koji Nakagawa, Nikolaj Popov, Judit Robu, Camelia and Markus Rosenkranz, Robert Vajda, Wolfgang Windsteiger, Alexander Zapletal (and I do hope I did not forget anyone, it's almost time to print this thesis). I did not forget, I just would like to especially thank: Tudor Jebelean for his support and advice both in RISC and after my return in Timişoara and for his efforts in setting up and keeping the e-Austria Institute going. Florina Piroi for help with the German translation, but not only (she would sit right across me in our office at RISC and have to face a barrage of my *Theorema* related questions, which she would always find time to answer, she is also really good friend). Laura Kovács, for her continuous support and friendship

(thanks also to Harald). Temur Kutsia helped me a lot with *Theorema*, always answered my (research) questions and always listened to my ramblings about revolution, life, the universe and everything else.

Thanks to all others at RISC, you made my stay there interesting. Carsten Schneider, thank you for helping with the German translation for this thesis, and all the rest through the years – we had some good times. Bogdan Mătăsaru (and Paici) thank you for taking care of me in those my early months in RISC. Cleo and Petru Pau, thank you so much – I abused your hospitality so many times, you have always been welcoming and supportive (I am almost like family ;)), and it would have been so much more difficult without your help.

Many thanks to Prof. Alan Bundy for pointing me to various approaches in program synthesis and his feedback of my work during my visit in Edinburgh. Also to the rest of the people I met in Edinburgh (in random order): Laura Meikle, Alison Pease, Colin Fraser, Lucas Dixon, Ewen Maclean, Roy McCasland and all the others (who I ask to forgive my weak memory of names). I learned a lot, and had a great time.

Many thanks to all my colleagues at the e-Austria Institute in Timişoara. Especially Cosmin Bonchiş, Cornel Izbaşa for "all that jazz" and Gabriel Istrate for his help and understanding. Thanks to those who worked with me in the SysteMaThEx project, especially Mădălina Hodorog for putting in the extra effort.

All my family has been tremendously supportive through these years. I owe so much to my parents. I hope they are proud how things (eventually) turned out. Mulţumesc!

Last but not least, this goes to Liana! A big big big big thank you for putting up with me all this time! I will make it up to you.

The work presented in this thesis was supported by an Upper Austrian Government grant, the Austrian Science Foundation (FWF) project SFB F1302, the Calculemus European Project, the European Project MERG-CT-2004-012718: SysteMaThEx.

Contents

List of Figures

CHAPTER 1

Introduction

In the motivation of his logic for computer science lecture notes [Buchberger, 1991], Bruno Buchberger describes *mathematics* as the technique of problem solving (*the dynamic aspect*) and obtaining knowledge (*the static aspects*) by reasoning. In this view, *reasoning* is the exploration of intellectual models.

Reasoning can be implemented on computers, for instance as mathematical assistant systems, to support *mathematicians* (i.e. those applying the method of mathematics) in problem solving. Negative results in *logic* (i.e. the science of reasoning) ensure that, in fact, such an implementation would never solve **all** problems but also that there will always exist potentially unlimited, interesting and novel ways to solve particular problems (and that the role of the mathematician is not reduced only to operate such a system).

In this introductory chapter, we give a short motivation for the two problems addressed by this thesis:

- *the problem of algorithm synthesis*, i.e. the problem of devising a method for the invention of algorithmic solutions to problems,
- *the problem of Gröbner bases*, an important subject in polynomial ideal theory, for which a successful application of a method for algorithm synthesis is non-trivial and, hence, interesting.

These problems are then placed in the broader context of mathematics, logic and computer science. Finally we give an overview of the structure of this thesis.

1.1 The Problem of Algorithm Synthesis

The goal of computer science is automation of problem solving.

Correctness of Algorithmic Solutions

Computers have become ubiquitous in everyday life, essential tools in many domains of human existence, and they enter new domains all the time. The number and complexity of programs grow. There is an ongoing effort to solve more and more problems, from various domains, with the help of computers.

The incorrectness of programs is an obvious concern and there are plenty of examples where programs caused problems in the "real world" (from simple annoyances to disasters, e.g. the Pentium processor division algorithm, Ariane 5 explosion, Mars Climate Orbiter loss, Therac-25 radiation machine). The reasons for incorrect programs can be multiple: from programming errors (i.e. incorrect implementation), to incorrect algorithms (wrong/incomplete solutions for the problems).

Several approaches are considered to address the correctness problem.

Testing algorithmic solution

Testing is the process of *examining* the behavior of programs, for individual input, with the aim to detect and eliminate errors. In fact, a great deal of effort is channelled towards testing of programs, and this process should detect incorrect implementations. (Note that programming language mistakes are not considered here — these should be detected and eliminated by the evaluation machinery, i.e. compiler, interpreter.) In principle, if the implementation is correct, but the program does not work as it is supposed to (detected by testing), then there must be something wrong with the algorithm. Program testing is very expensive - it amounts to checking as many as possible instances of the argument. In many cases, exhaustive testing is not possible, or at least not viable. Due to these considerations, testing does not, in fact, ensure correctness.

Verifying Algorithmic Solutions

Another way is to try to prove that the programs are correct, i.e. *formal verification of programs*. The advantage of this approach is that a correctness proof represents a guarantee for program correctness, ideally making testing unnecessary. However, in case the correctness proof fails, it is up to the designer of the algorithm to correct the solution.

There is also an issue about the language level where the verification is done: If the verification is not done on the program itself, but on a higher level, in a language where the reasoning rules used in the proof can be applied, then the algorithm (solution to the problem) may be proven correct, but there may still be errors in the implementation.

Inventing Correct Algorithmic Solutions

Yet another way is to generate a correct solution to the problem, by starting from the *problem specification* and following a number of derivation steps (reasoning steps). This is *algorithm synthesis*. If an algorithmic solution can be formally derived starting from a problem specification, then this solution will be guaranteed to be correct, by the construction (modulo the assumption that the derivation mechanism is sound). Thus, ideally, program synthesis brings more to the correctness of programs than just verification.

Here still, if the output of synthesis is an algorithm expressed in a higher level (abstract) language (suitable for the derivation steps necessary in the synthesis process), and the implementation itself is in a lower level (machine) language then, although the algorithmic solution is correct, the correctness of the implementation is still an issue.

If, however, both the specification and the solution algorithm (program) are expressed in *one language frame*, then the synthesis process provides the guarantee that the algorithm is correct.

One possible language frame for synthesis is *predicate logic*. A fragment of predicate logic (restricted to universally quantified formulae, with finite ranges, and recursion, based on substitution and replacement), can be used as a programming language, and the *Theorema* system (which we use to investigate the case studies presented in this thesis) uses a version of predicate logic as its language. Mathematics can be formulated in predicate logic, therefore a synthesis mechanism for predicate logic can provide algorithmic solutions to mathematical problems.

Lazy Thinking: A Solution to the Problem of Algorithm Synthesis

The *lazy thinking method* for algorithm synthesis, first proposed by Bruno Buchberger in [Buchberger, 2003], is a solution to the problem of algorithm synthesis in the frame of predicate logic.

The specification for an algorithm is a *predicate logic formula*, i.e. a statement of the correctness theorem for the desired algorithm. Lazy thinking proposes the following methodology to obtain the algorithm that solves the problem:

- Start from a "synthesis situation": a *specification of* an algorithm, together with the *knowledge* necessary for the formulation of this specification.
- Choose an *algorithm scheme* from a library of schemes, i.e. "concentrated algorithmic knowledge". An algorithm scheme is a definition of the new algorithm, expressed in terms of *unknown subalgorithms* (i.e. algorithms that are new to the theory, and for which no definitions are available). Add this definition to the knowledge base.
- With this knowledge, attempt to prove the correctness theorem. This will likely fail, due to the unknown subalgorithms introduced by the algorithm scheme.

- Analyze the failing proof situation and from the failure, generate conjectures on the unknown subalgorithms that will allow the proof to get over the failing proof situation.
- Repeat the previous step until the proof is completed.

If this procedure stops, the result of lazy thinking is a proof of the correctness of the proposed solution, together with a set of conjectures generated during the process. These conjectures — on the unknown subalgorithms — can be considered as their specifications. Thus, the initial synthesis problem is reduced to the problem of finding subalgorithms that satisfy the generated specifications. Now, either retrieve suitable subalgorithms from the knowledge base, or apply lazy thinking again for synthesizing them.

1.2 Gröbner Bases

Brief Description of the Method

The method of Gröbner bases, an algorithmic method in multivariate polynomial ring theory, was introduced by Bruno Buchberger in his PhD thesis, [Buchberger, 1965].

Consider a multivariate polynomial ring over a field. A *polynomial* p can be seen as a set of pairs of *power products* and their corresponding nonzero *coefficients* from the coefficient field. On the power products an *admissible ordering* is introduced: it is a total ordering such that (the power product) 1 is the smallest element, and this ordering is monotone w.r.t. multiplication. This induces a well-founded partial ordering on the multivariate polynomial ring.

Fix such a polynomial ordering. A relation (and corresponding operation) of *reduction modulo a polynomial set* can be defined on the multivariate polynomial ring, such that the reduced elements are smaller than the element being reduced. *Total reduction modulo a polynomial set* can be defined as repeated iterations of the reduction operation (until no reduction is possible), and it produces a *normal form* of polynomials w.r.t. the reduction relation.

A *Gröbner basis* is a polynomial set G for which the reduction modulo G has the Church-Rosser property, i.e. reducing a polynomial modulo two distinct polynomials, then totally reducing the resulting elements, gives the same result. For any polynomial set F, there exists a Gröbner basis G such that the transitive reflexive closure of the reduction relation F and G are equal (which is equivalent to saying that the congruence relations modulo F and G are equal).

The breakthrough of [Buchberger, 1965] consisted in finding a Gröbner basis algorithmically: It is sufficient to check finitely many "critical situations" ([Buchberger, 1998]) to see whether the Church-Rosser property holds. If not, then the set F is augmented with the "S-polynomial" that "injures" this property. When no more such situations are detected, the polynomial set is a Gröbner base.

Importance

The method of Gröbner bases has numerous applications: polynomial ideal problems (ideal membership, ideal inclusion), solving systems of polynomial equations, etc. Gröbner bases proved useful in many domains: algebraic geometry, functional analysis, coding theory, cryptography, program verification, symbolic summation, theorem proving, combinatorics, graph theory, etc. The algorithm is implemented in many computer algebra systems, such as *Mathematica, Maple, CoCoa, Macaulay, Singular*, etc.

There are numerous articles on the subject of Gröbner bases (600+), several (tens of) textbooks, many computer algebra and symbolic computation research groups address this subject, it recently (2006) had a special semester dedicated to it in Linz, Austria, and it has its own entry in the MSC classification of mathematics: 13P10.

Further Reading

For introductions to the method, see Bruno Buchberger's papers [Buchberger, 1985], [Buchberger, 1998], also the textbooks [Winkler, 1996], [Cox et al., 2005], or [Buchberger and Winkler, 1998] (the latter containing detailed descriptions of the many applications of the method), or [Grabmeier et al., 2003] (for a brief overview with many pointers).

A Gröbner bases bibliography is available online, see [Buchberger and Zapletal, 2006].

1.3 Context

We have, so far, motivated why the two problems we address in this thesis are important. The suggestion is, for the problem of algorithm synthesis, that a one-language frame approach, which uses the same language for the algorithm being discovered and the discovery process, will have advantages. Predicate logic is a universal language (it is, in particular, the language of mathematics), Gröbner bases is a very important algorithmic method and the natural question is whether the algorithm for Gröbner bases can be synthesized, in predicate logic, automatically.

In this section, we give an overview of the broader context: logic, mathematics, their implementation on computers and recent trends in using computers to do mathematics (in a style close to what mathematicians do) We also identify the place of our work in this context. The presentation of the context is not exhaustive but focuses on the material relevant for our subject.

Historical Context

Gottfried Wilhelm Leibniz envisioned a "universal calculus", to go beyond the power of syllogistic logic (the state of the art in logic at the time), that would determine which logical inferences are logically valid, and was looking for a "universal characteristic" that would allow him to apply this calculus to establish the truth of scientific propositions by "calculating" them ("Calculemus!"), see [Lenzen, 2004].

Gottlob Frege introduces in his *Begriffsschrift* [Frege, 1879] the first formal description of predicate logic (his was a second order system), which he uses to define mathematical concepts and to prove interesting propositions about them. He further develops his system in *Grundgesetze der Arithmetik* [Frege, 1893], [Frege, 1903], and tries to derive number theoretical results. However this extension proved inconsistent, as Russel pointed out by his famous paradox.

Bertrand Russel and Alfred North Whitehead published *Principia Mathematica*, in 3 volumes: [Whitehead and Russel, 1910], [Whitehead and Russel, 1912], [Whitehead and Russel, 1913]. In this highly influential work, the authors set out to provide a logical apparatus for the development of mathematics, which should avoid paradoxes such as Russel's, that made Frege's system inconsistent. Although it was controversial whether the goal of *Principia Mathematica* was achieved (not all the axioms were accepted), the 3 volumes did contain the development of significant parts of mathematics and suggested that the project is feasible.

David Hilbert proposed as his second problem a direct proof of the consistency of analysis, in his lecture at the International Congress of Mathematicians, in 1900 [Hilbert, 1900], for whose solution he needed a logical apparatus not available at the time. Following the progress achieved by *Principia Mathematica*, he returned to work on foundational problems. In the beginning of the 1920's, he formulates a proposal, known as Hilbert's Program, in which he calls for an axiomatization of all mathematics together with a "finitary" proof that this axiomatization is consistent. Hilbert and his collaborators (Paul Bernays, Wilhelm Ackermann, John von Neumann, Jacques Herbrand) made great progress on the Program in the 1920's, even claiming success at certain moments. However, soon afterwards, incompleteness results spelled the end for the program in its original form.

Kurt Gödel, in his incompleteness theorems (1930) [Gödel, 1931], proved that, in fact, Hilbert's Program could not be carried out, i.e.

- a recursively enumerable axiomatization of arithmetic, or of any theory in which arithmetic can be embedded, is impossible (the first incompleteness theorem), and
- one cannot prove the consistency of a theory (containing arithmetic) within that theory (the second incompleteness theorem).

Moreover, in 1936, Alonzo Church, see [Church, 1936] and Alan Turing, see [Turing, 1936], discover independently that full first order predicate logic is undecidable, i.e. there is no mechanical way (algorithm) that can be used decide the truth of statements in this

logic.

Although the negative results mentioned here dealt a heavy blow to the logicist approach (Hilbert's Program), they present a new challenge: what are those fragments of mathematics/theories that are decidable. Several were identified: Presburger arithmetic, real closed fields, abelian groups, linear orderings.

Another aspect concerning these important results was that their formulation and proofs used notions about computable functions, i.e. algorithms. The computer age began just a few years later.

Logic and Computation

Apart from the (unfortunate) military factor and the engineering applications, the one force driving the development of computers in the early days of the computer age was mathematics (see, for example, [Tanenbaum, 2006] for a brief history). Very soon people started using computers to investigate logic and mathematics through logic. This included the decidable theories mentioned above, implementation of calculi and decision procedures, etc.

In 1954, Martin Davis programed a decision algorithm for Presburger arithmetic. This was, however, very inefficient.

Newell, Shaw and Simon, see [Newell et al., 1995], implemented the Logic Theory Machine to carry out proofs in the propositional calculus of *Principia Mathematica*. The emphasis is on *heuristics* versus the exhaustion of the search space (giving up completeness for efficiency).

The investigation of first order logic led to several developments, such as the Davis–Putnam procedure for formula satisfiablity, see [Davis and Putnam, 1960]. However, the breakthrough was achieved with the introduction of the resolution rule by John Alan Robinson, see [Robinson, 1965]. This is considered by many as the birth of automated theorem proving. Resolution is complete for first order logic, and was easy to implement. For an overview, with more details, of the early developments, see [Davis, 2001].

Not only were computers used to investigate logic, but logic was used to do computation. John McCarthy proposed in 1958, see [McCarthy, 1958], the use of mathematical logic as a base for programming:

> "... programs to manipulate in a suitable formal language (most likely a part of the predicate calculus) common instrumental statements. The basic program will draw immediate conclusions from a list of premises. These conclusions will be either declarative or imperative sentences. When an imperative sentence is deduced the program takes a corresponding action."

Logic programming has caught on and developed as a well-established field. Its main applications are expert systems and automated theorem proving. Well-known logic pro-

gramming languages are Planner, Prolog.

Other logical formalisms, such as the λ–calculus (introduced by Church), and combinatory logic (Schönfinkel, [Schönfinkel, 1967], and Curry, [Curry, 1942]), were used as a basis for another programming paradigm, *functional programming*. Well–known functional programming languages are Lisp, ML, Haskell. This programming paradigm has numerous applications, both scientific and industrial.

Computer Mathematics

The advent of computers had a significant impact on the way mathematics is done. In fact, at the moment a mathematician could hardly imagine working without the computer (writing papers, communicating with others, even using computers to carry out typical mathematical activities).

The basic mathematical activities include exploration of mathematical theories through *defining notions*, *formulating* and *proving* properties of these notions, *computing* (or *simplifying*) based on these notions, *formulating problems* and *finding solutions* (*solving*), as well as writing ones findings in various forms.

Proving: Computer Supported Formalization of Mathematics

Using computers to formulate mathematical theories and check that the formulation is correct was first attempted by the Automath group led by de Brujin, from 1967 to the early 1980's, see [Nederpelt et al., 1994]. The goal was to *formalize a large portion of mathematics* (formulate the notions, properties of the notions, and proofs - an analysis book was formalized), and *check* it using a small program that is easy to prove correct (by hand). This idea, of a small proof kernel that should check the correctness of proofs (produced either by humans, by formalization, or semi/automatically by reasoning systems) was adopted by a large number of subsequent systems.

Mizar [Rudnicki, 1992] is another notable proof checking system. Started in 1973, it is "an attempt to reconstruct [a] mathematical a vernacular in a computer oriented environment" (`http://mizar.uwb.edu.pl/project/`). Articles (i.e. fragments of mathematical theories) formalized in the Mizar language are checked by the system, and added to the Mizar Mathematical Library (MML) - one of the biggest computer processed mathematical knowledge bases available.

Proving: Computer Supported

Other reasoning systems aim to provide assistance for constructing proofs (interactive theorem provers). Such systems are ACL2 [Kaufmann et al., 2000], Coq [Bertot and Castéran, 2004], HOL [Gordon and Melham, 1993], IMPS [Farmer et al., 1996], Isabelle [Nipkow et al., 2002], Nuprl [Constable et al., 1986].

Alan Bundy introduces in [Bundy, 1988] the idea of *proof planning*: first construct a plan for a proof, then use it to guide the construction of a proof. Plans represent common proof patterns, and combinations thereof. Oyster/Clam [Bundy et al., 1990], and Omega [Benzmller et al., 1997] are examples of reasoning systems based on proof planning.

The *Theorema* system, see [Buchberger et al., 1997], [Buchberger et al., 2000], [Buchberger et al., 2006] for details, is a system that aims at providing support (in the form of natural style provers, simplifiers and solvers) for doing mathematics.

Solving and Computing

Computer algebra systems provide algorithms for solving and computing using symbolic methods, from [Grabmeier et al., 2003]:

> "Computer Algebra is a subject of science devoted to methods for solving mathematically formulated problems by symbolic algorithms, and the implementation of these algorithms in software and hardware."

Computer algebra systems are capable of solving problems in various theories, such as arithmetic, linear algebra, polynomials, group theory, summation, integration, differential equations, etc. Some of the best known are Axiom, Aldor, Macsyma, Magma, Maple, Mathematica [Wolfram, 2003], Reduce, etc. (general purpose), CASA, CoCoa, Singular, etc. (special purpose). We point the reader to [Grabmeier et al., 2003], which contains short descriptions of all the systems mentioned (and more), with further references.

Numerical analysis. So far, we mentioned symbolic aspects of computation (logic, theorem proving, computer algebra). However, a very significant field of study, numerical analysis uses numerical computations to solve problems in continuous mathematics. Subjects of interest are linear algebra, solving of systems of equations, differential equations, etc, see [Duff and Watson, 1998], for a survey. Numerical analysis has many applications in engineering, medicine (e.g. tomography), etc.

Bringing it All Together: Mathematical Knowledge Management

While computer mathematics has seen tremendous developments in the last few decades, there is a lack of integration of the various tools available and lack of interest from "traditional mathematicians".

Mathematical Knowledge Management(MKM) is a new research field, established by the First International Workshop on Mathematical Knowledge Management, in 2001, see [Buchberger and Caprotti, 2001], which tries to bring together and focus all the efforts to reach this integration, i.e. to provide tools that allow efficient management of mathematical knowledge.

Bruno Buchberger proposed in [Buchberger and Nakagawa, 2004] the following breakdown of the field of MKM:

(a) the *organizatorial aspect* of MKM, which is concerned with:
- "digitization of mathematics": scanning of the large amounts of mathematical knowledge available in print into electronic formats (ps, pdf), the transformation of these into representation formats that allow editing and publishing (e.g. TeX [Knuth, 1984], LaTeX [Lamport, 1986]);
- parsing of LaTeX documents, detecting the logical structure of documents, see [Nakagawa et al., 2004];
- providing standards for representation of mathematical knowledge (with its logical structure), such as MathML, OpenMath, see [Carlisle, 2000];
- providing tools for the organization of mathematical knowledge, user interfaces, etc., e.g. see [Piroi, 2004];

(b) the *logical aspect* of MKM, which is at the core of the field, and which is concerned with:
- formalization of mathematics, i.e. transferring mathematical knowledge into some electronic, formally checked format, which was started by the Automath project [Nederpelt et al., 1994], continued with Mizar [Rudnicki, 1992], and which at some level or the other is found in the context of every theorem prover, proof assistant;
- methods to ensure communication between various formalisms used in various systems (e.g. logical frameworks [Pfenning, 1996]);
- (computer supported) exploration of mathematical theories, invention of mathematics, advocated by Bruno Buchberger and the *Theorema* group, see [Buchberger, 2004a], [Buchberger et al., 2006].

Summary and Placement in Context

In the above, we tried to give an overview of the interplay of computers, logic and mathematics. We mentioned a few of the milestones of this interplay. The progress made in the last few decades in all these domains allows them to interact more closely and naturally, and this interaction is likely to be stronger and stronger. In particular, one of the main goals of MKM is to do "serious" mathematics by computers using a language familiar to the working mathematician - (a form of) predicate logic: this is the *theory exploration aspect* of MKM.

It is here where the work presented in this thesis is localized. Synthesis of algorithms is seen as invention of new concepts (algorithmic functions) in mathematical theories, and algorithms are expressed in predicate logic. We undertake the synthesis of an algorithm for Gröbner bases, an important and nontrivial (as discussed above) case study, as a benchmark for the synthesis method.

1.4 Goals and Achievements of this Thesis

The major goals of the work described in this thesis are:

- the *implementation of the lazy thinking method for algorithm synthesis*, in the context of systematic theory exploration and invention,
- the application of the synthesis method to nontrivial case studies, in particular to the *invention of an algorithm for Gröbner bases computation.*

These goals were addressed in the following manner:

Implementation

We implemented, in *Theorema*, the *lazy thinking method* for algorithm synthesis. This consisted in the implementation of:

- the *"cascade"* i.e. the loop over the proof attempts,
- the *failing proofs analyzer*,
- and the *conjecture generator* from failed proof attempts.

The implementation of lazy thinking was applied to various synthesis examples, including the synthesis of sorting algorithms in the theory of tuples.

This implementation is a contribution to the theory exploration capability and support of the *Theorema* system. It provides a flexible and powerful tool to be used in a future framework for theory exploration in *Theorema*.

Case study: Synthesis of a Gröbner Bases Algorithm

We carried out in *Theorema* the case study of Gröbner bases algorithm synthesis, following the outline proposed by Bruno Buchberger in [Buchberger, 2004b]:

- formulate the problem of Gröbner bases, and the corresponding knowledge base,
- formulate and implement special *Theorema* provers to reason about the problem at hand,
- formulate the Critical-Pair Completion algorithm scheme, formulate and implement an induction principle used to derive propertied of this scheme, and then
- set up the lazy thinking cascade and obtain (using our implementation) natural specifications for the subalgorithms used in the Critical-Pair Completion scheme, which, in fact, leads to the invention of the Buchberger algorithm.

This case study shows that the lazy thinking method is powerful enough to address "real" mathematical problems - in this case study we show that the method manages to synthesize automatically the essential ideas of Gröbner bases (S-polynomials) that were invented by Bruno Buchberger by thinking about the specific problem of Gröbner bases construction, when he introduced the concept.

1.5 Statement of Originality

The work presented in this thesis was carried out in the frame of the *Theorema* project at the Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Theorema is developed under the guidance of Bruno Buchberger, who set out from the start to design a system that provides support to the working mathematicians, in the sense already discussed in Section 1.3 of this Chapter where these ideas are placed in the context of (computer supported) mathematics. The development of the system and the work related to this development are consistent with Bruno Buchberger's vision of how mathematics can (and should) be done.

Therefore, in particular, credit goes to Bruno Buchberger for the ideas on which this work is based:

- *systematic theory exploration* (see [Buchberger, 2004a], [Buchberger et al., 2006]), and in particular,
- the *lazy thinking method for algorithm synthesis* (introduced in [Buchberger, 2003]),
- the general outline of proof failure analysis and conjecture generation,
- the use of the lazy thinking method for synthesizing Gröbner bases algorithms (see [Buchberger, 2004b]).

The author's own contribution to this thesis consist of:

- the adaptation of the lazy thinking method for a particular view of Buchberger's theory exploration model,
- implementation in *Theorema* of the lazy thinking method (lazy thinking cascade, failure analysis and conjecture generation),
- application of the implementation in case studies (including synthesis in tuple theory, natural numbers, Gröbner bases theory),
- development and implementation of various provers needed to handle the case studies.

During the course of the work described in this thesis, several technical problems had to be overcome.

1.6 Structure of the Thesis

Chapter 2 is dedicated to the description of the lazy thinking synthesis method. We first give a short overview of the scheme based theory exploration model in which lazy thinking is applied. Then we describe the lazy thinking method for synthesis in this context: algorithm schemes, proof methods corresponding to algorithm schemes, failing proof analysis and conjecture generation. Several examples illustrate the application of the method for the synthesis of algorithms, but also for the synthesis of "interesting"

constants, showing the versatility of lazy thinking for theory exploration. Also illustrated is the situation when the specification cannot be satisfied.

In Chapter 3 we present the implementation of the lazy thinking method in *Theorema*. Some background information of the *Theorema* system, relevant to our implementation, is given first. Then we describe in detail the implementation of the lazy thinking cascade, the failure analyzer and the conjecture generator. We provide descriptions for the user, an informal presentation of each of the components of our implementation, as well as the code.

The case study of Gröbner synthesis is presented in Chapter 4. We describe the setting of the case study: knowledge on polynomials, reduction relations, polynomial reduction modulo a polynomial set. Then we give the specification for the Gröbner bases algorithm: we want an algorithm that given a polynomial set as an input, returns a Gröbner basis, i.e. such that the reduction modulo the result has the Church-Rosser property, and the ideal generated by the result and that generated by the argument are the same. The Critical-Pair/Completion algorithm scheme is proposed as a solution. To carry out the synthesis process, several difficulties (due to the algorithm scheme) have to be overcome: the proof methods attached to the algorithm schemes, preprocessing of the algorithm scheme, termination.

In Chapter 5 we present related work: first a short description of work about invention in mathematics, followed by an overview of synthesis work and finally formal Gröbner bases theory.

Chapter 6 presents the conclusions of the work and points to future developments.

CHAPTER 2

The Method of Lazy Thinking

Lazy thinking is a scheme-based deductive algorithm synthesis method in predicate logic, which plays an important role in a scheme-based systematic theory exploration model proposed by Bruno Buchberger, see [Buchberger, 2004a], [Buchberger et al., 2006]. This chapter is dedicated to the presentation of the method, together with examples of how it is applied, taken from the exploration of various theories.

2.1 Scheme-Based Mathematical Theory Exploration

Computer supported, systematic mathematical theory exploration represents, in this author's view, a new way of doing mathematics. The advantages are multiple: it can stimulate rigor (the computer cannot be convinced by hand-waving arguments), discipline (formalism), awareness (the tools and methods of mathematics are explicitly available to the mathematician, and not hidden in implicit reasoning habits). It can help mathematicians make progress in their research by taking off their hands the routine proving, calculations, solving. It can teach a student how to do mathematics, and how to apply the method of mathematics in other domains.

Bruno Buchberger has, since many years, been advocating this idea. *Theorema* is a mathematical assistance system that was set up from the beginning to support the exploration of mathematical activities. The papers [Buchberger, 2004a], [Buchberger et al., 2006] describe these ideas, and point to earlier relevant papers.

This section contains the author's interpretation of Buchberger's theory exploration model. The following are discussed: the language of a theory, what is a theory, knowledge

schemes and steps of theory exploration in this model.

The Language

The language is a version of predicate logic (like the *Theorema* language):

- First order logic with equality, together with the associated inference rules are used to express *mathematical facts*, and reason about them. The language is untyped, and (unary) predicates are used to express type information for objects. A fragment of this language can be used for programming: universally quantified formulae, with finite ranges, and recursion, using substitution and replacement as computational engines.
- Higher order logic (with the restriction that higher order variables appear only universally quantified) are used to express *mathematical ideas* (or *schemes*). The inference rule needed to reason about the formulae expressed in this fragment of the language is "arbitrary but fixed".
- Sequence variables (i.e. variables that can be instantiated with a finite sequence of terms) are also allowed in the language (together with the corresponding inference rules - for sequence equality and sequence induction). Predicate logic with sequence variables is described in [Kutsia and Buchberger, 2004].

Mathematical Theories

A mathematical theory $\mathcal{T}$ is defined by its *(first order) language*, its *knowledge base* (collection of formulae), and its *reasoning mechanism*:

- The *first order language* $\mathcal{L}$ describing the theory is a triple: $\mathcal{L} = \left\langle \mathcal{P}, \mathcal{F}, \mathcal{C} \right\rangle$, where $\mathcal{P}$ is the set of predicate symbols (including the binary equality "=" predicate, and one or more unary predicates describing the "type" of objects in a theory), $\mathcal{F}$ is the set of function symbols (including the unary identity "*id*" function), $\mathcal{C}$ is the set of constants of the theory. For practical reasons, we distinguish between function symbols and constants, although the latter can be seen as 0–ary functions.
- The *knowledge base* $\mathcal{KB}$ of the theory consists of a collection of formulae over the language $\mathcal{L}$. It consists of the collection of axioms and theorems of the theory.
- The *reasoning mechanism* $\mathcal{IR}$ of the theory contains all reasoners available to the one developing the theory. This includes, for any theory, the first order predicate logic calculus and rewriting (since predicate logic with equality is our language frame). In addition, specific (i.e. theory dependent) reasoning rules.

Knowledge Schemes

Knowledge schemes (ideas) are higher order formulae that describe how certain notions (functions, predicates) relate to other notions. Schemes can be considered as "condensed experience" of mathematicians for constructing mathematical knowledge. Schemes are stored in libraries of schemes, available to those developing mathematical theories.

Knowledge schemes are available at various levels of abstraction: some are general, while others are special(ized),i.e. developed in the frame of special theories.

Example 1 (General schemes: algebraic structures)**.** Some examples[1] of independent schemes are, for instance those denoting algebraic structures:

$$\underset{p,op}{\forall}(\textit{is-semigroup}[p,op] \Leftrightarrow \underset{p[x,y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} p[op[x,y]] \\ op[x,op[y,z]] = op[op[x,y],z] \end{array} \right) ,$$

$$\underset{p,op,zero}{\forall}(\textit{is-monoid}[p,op,zero] \Leftrightarrow \underset{p[x,y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} \textit{is-semigroup}[p,op] \\ op[x,zero] = x \end{array} \right) ,$$

$$\underset{p,op,zero,inv}{\forall}(\textit{is-group}[p,op,zero,inv] \Leftrightarrow \underset{p[x]}{\forall} \bigwedge \left\{ \begin{array}{l} \textit{is-monoid}[p,op,zero] \\ op[x,inv[x]] = zero \end{array} \right) .$$

In the formulae above, $p, op, zero, inv$ are higher order variables, and *is-semigroup*, *is–monoid*, *is–group* are special higher order constants (names, unique identifiers of schemes).

Example 2 (General schemes: relation structures)**.** Other general schemes are relation structures, like:

$$\underset{p,r}{\forall}(\textit{is-preorder}[p,r] \Leftrightarrow \underset{p[x,y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} r[x,x] \\ (r[x,y] \wedge r[y,z]) \Rightarrow r[x,z] \end{array} \right) .$$

$$\underset{p,r}{\forall}(\textit{is-partial-ordering}[p,r] \Leftrightarrow \underset{p[x,y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} \textit{is-preorder}[p,r] \\ (r[x,y] \wedge r[y,x]) \Rightarrow x = y \end{array} \right) .$$

Example 3 (Special, theory-dependent schemes)**.** Consider now the theory of natural numbers, with the constant 0, successor function $^+$, and predicate *is-nat* in the language of the theory (see details in [Hodorog and Crăciun, 2007]). The following are natural numbers knowledge schemes:

$$\underset{f,g,h}{\forall}(\textit{is-rec-nat-binary-fct-1r}[f,g,h] \Leftrightarrow \underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x,0] = g[x] \\ f[x,y^+] = h[f[x,y]] \end{array} \right) ,$$

$$\underset{f,g,h}{\forall}(\textit{is-rec-nat-binary-fct-1l}[f,g,h] \Leftrightarrow \underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \left\{ \begin{array}{l} f[0,y] = g[y] \\ f[x^+,y] = h[f[x,y]] \end{array} \right) ,$$

$$\underset{f,g,h}{\forall}(\textit{is-nat-rec-binary-rel-2}[f,g,h] \Leftrightarrow \underset{\textit{is-nat}[x,y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x,0] \Leftrightarrow g[x] \\ f[x,y^+] \Leftrightarrow (h[x,y] \vee f[x,y]) \end{array} \right) .$$

[1] We use *Theorema* notation: If p is a unary predicate, $\underset{p[x,y,z]}{\forall} \ldots$ abbreviates $\underset{p[x]}{\forall}\underset{p[y]}{\forall}\underset{p[z]}{\forall} \ldots$

The recursive structure of the above scheme reflects the inductive structure of natural numbers.

In a similar fashion, given an inductive domain, one can construct systematically recursive schemes based on the inductive structure.

Concerning the *structure of a knowledge scheme*, let us consider one of the schemes in the above examples:

$$\underbrace{\underset{f,g,h}{\forall}}_{(1)} \underbrace{(\textit{is-rec-nat-binary-fct-1r}[f,g,h]}_{(2)} \Leftrightarrow \underbrace{\underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x,0] = g[x] \\ f[x,y^+] = h[f[x,y]] \end{array} \right.}_{(3)}).$$

A knowledge scheme is:

- universally quantified over higher order variables that stand for concepts in first order theories (functions, predicates, constants), see (1);
- it is an equivalence (definition), with the left hand side a higher order predicate applied to the concept variables, see (2): the name of the higher order predicate is unique, and it can be seen as a label which can be processed in the frame of the language;
- the right hand side of the equivalence (3) is formulated in the first order theory in which the concept variables are considered as constants.

To summarize, schemes are added to libraries based on experience (interestingness) and structure. While some of them are abstractions of mathematical ideas that were crystallized over years and years of doing mathematics (or writing programs). Alternatively, some of them, dependent on recursive theories, can be generated systematically based on the structure of the theory.

Exploration

The goal of theory exploration is the development of mathematical theories. An *exploration situation* consists of a theory being developed, together with a library of knowledge schemes, representing, as mentioned above, accumulated mathematical ideas available to the explorer. These schemes play an essential role in the exploration.

We will now present the basic steps that can be taken in the development of a theory $\mathcal{T} = \langle \mathcal{L}, \mathcal{KB}, \mathcal{IR} \rangle$:

Add propositions to the knowledge base. Theories are usually introduced by a set of axioms. A first step in their development is adding to $\mathcal{KB}$ "interesting, useful" consequences of these. Strategies for obtaining potential statements of consequences can be employed: modify the statements of the axioms by exploiting the structure of the axioms (e.g. if an axiom about left neutral element is present in $\mathcal{KB}$, try to

prove the right neutral property). Alternatively, consequences can be derived by forward reasoning. The aim of this type of exploration round is to "saturate" $\mathcal{KB}$, i.e. to add "all interesting" consequences of an initial knowledge base. This may be, however, an incomplete process. Producing all consequences can be costly, lead to an explosion in the number of propositions, and many of the propositions thus produced would not be interesting for a human mathematician.

Add concepts to the language and explore their properties. *Concepts* in mathematical theories are functions, predicates or constants. Usually, mathematical theories are augmented with the introduction of new functions or predicates, by definition. To do so, *definition schemes* can be instantiated with a new symbol (which is added to the language $\mathcal{L}$), for the concept being defined, and with symbols from $\mathcal{L}$ corresponding to the concepts used to define the new concept.

Following the introduction of the new concepts, its properties are investigated:

- look for equivalent definitions (again trying out modifications of form of the definition);
- investigate structural properties of the concept (e.g. whether it is commutative, associative, etc. for binary functions, or whether we have an equivalence or an ordering for binary predicates, etc.). The potential structural properties can be obtained by instantiation of *proposition schemes*;
- investigate how the new concept interacts with other concepts in the theory, with proposals for such properties again obtained from instantiations of schemes.

Again, the goal is a kind of "saturation of the theory".

Solve problems. *Problems* are situation where a *solution* is desired. They can be introduced in the theory by instantiations of schemes.

Example 4 (Inverse of natural numbers)**.** Consider, in the theory of natural numbers, with *is-nat*, 0, + in the language, introduced in the usual way, the following problem: *is-group*[*is-nat*, +, 0, $\ominus$], i.e. we have a situation where we desire a new concept, denoted by $\ominus$, such that the folloging holds:

$$\underset{\textit{is-nat}[x]}{\forall} \bigwedge \left\{ \begin{array}{l} \textit{is-monoid}[\textit{is-nat}, +, 0] \\ x + \ominus x = 0 \end{array} \right. .$$

There are two ways to solve problems:

- *search for solutions* in $\mathcal{KB}$ (knowledge retrieval), i.e. find concepts that satisfy the specification of the problem;
- *invent a solution* that satisfies the problem; a method for doing so is *lazy thinking*, described in detail in the following Section 2.2.

As a result of the exploration round of solving a problem, $\mathcal{KB}$ (and $\mathcal{L}$, if solutions are invented) are updated.

Add new inference rules to the inference mechanism. During the development of the theory, it is possible to add new (theory dependent) inference rules to $\mathcal{IR}$, once certain knowledge is available. The application of these rules will result in shorter proofs. One way to add new inference rules to $\mathcal{IR}$ is to *lift knowledge to the level of inference*. This is a process very common in mathematics. One situation in which this can be applied is when the axiom system contains axiom schemes that cannot be represented in first order logic. Other examples include the introduction of new induction rules once well-founded (partial) orderings are available in the theory (as described in [Hodorog and Crăciun, 2007]).

As a result of lifting knowledge to inference, $\mathcal{IR}$ and $\mathcal{KB}$ are updated.

The overview on theory exploration given above is the author's interpretation and development of ideas formulated by Bruno Buchberger in [Buchberger, 2004a], [Buchberger et al., 2006]. Some examples of theory exploration steps carried out in the scheme based model are presented in [Hodorog and Crăciun, 2007].

2.2 Lazy Thinking for Algorithm Synthesis

We now describe in more detail lazy thinking, a scheme-based method for synthesis in the context of theory exploration.

2.2.1 Synthesis Situation

As mentioned in the previous section, synthesis is a theory exploration step, that transforms the theory with the statement of a problem into a theory with a solution to the problem. Therefore, a *synthesis situation* consists of:

- *The statement of the problem*, i.e. a formula of the form:

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

  where d is a predicate indicating the "type" of the variable x, and P is a formula (predicate) expressing a relation between the object x and the solution $A[x]$. The symbols d, and those in P are part of the language $\mathcal{L}$, while A is the symbol denoting the new concept - the intended solution of the problem. Note also that P can be arbitrarily complex. The statement of the problem, is, in fact, the correctness theorem for the desired algorithmic solution, i.e. the statement that A solves the problem P. (Here we confine ourselves to "explicit" problems, i.e. problem specifications of the form given above, which in fact covers most of the problems discussed in computer science.)
- *The theory corresponding to the problem*, which contains all definitions and properties of the symbols used in the statement of the problem. In other words, we want a solution to a problem that is well understood.

Part of the theory corresponding to the problem is the inference mechanism available. Typically, the problems we want to solve (algorithmically) are formulated in inductive domains (i.e. d describes an inductive domain). Several induction rules may be available, with the selection of one or the other determining the outcome of the synthesis process.

- *A library of algorithm schemes* (algorithmic ideas), from which it is possible to select a proposal for the desired solution. Attached to the scheme is information on the reasoning method (induction type) to be used in reasoning about the solution, and possibly other information (e.g. type information for subalgorithms).

2.2.2 Algorithm Schemes

Algorithm schemes are higher order formulae, that capture algorithmic ideas.

The *structure of an algorithm scheme* is similar to that of other schemes, but a few comments are in order. In fact, the scheme presented in Section 2.1:

$$\underbrace{\underset{f,g,h}{\forall}}_{(1)}\ (\underbrace{\textit{is-rec-nat-binary-fct-1r}[f,g,h]}_{(2)} \Leftrightarrow \underbrace{\underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x,0] = g[x] \\ f[x,y^+] = h[f[x,y]] \end{array} \right.}_{(3)})$$

can be used as an algorithm scheme. The following considerations will be taken into account when it comes to algorithm schemes:

(1) Among the higher order variables, one stands for the symbol being defined (f above). The others are *auxiliary subalgorithms* (*unknown functions, predicates, constants*).
(2) The use of the left hand side of the equivalence is the same as with other types of schemes.
(3) The right hand side of the equivalence is a (typically recursive) definition of the function (algorithm) in terms of the unknown subalgorithms. The structure of the recursion determines the structure of the induction rule to be used in reasoning about the algorithm scheme.

Building Libraries of Algorithm Schemes

There are two (possibly overlapping) sources from which algorithm schemes can be added to the library of schemes:

- systematic construction of recursive definitions in an inductive domain, or,
- algorithmic experience (the accumulated experience of computer mathematicians),

illustrated in the following examples.

Example 5 (Systematic build-up of recursive schemes for natural numbers)**.** The first order theory of natural numbers is introduced axiomatically in the usual way, see, for instance [Hodorog and Crăciun, 2007]:

- The language: $\mathcal{L}_{\mathbb{N}} = \left\langle \langle \textit{is-nat}, = \rangle, \langle {}^{+}, id \rangle, \langle 0 \rangle \right\rangle$, where *is-nat* is the unary predicate symbol that characterizes natural numbers, $^{+}$ is an unary function symbol (the successor function), the identity and equality symbols (unary and binary, respectively).
- The knowledge base $\mathcal{KB}_{\mathbb{N}}$, consisting of axioms for equality of natural numbers (reflexivity, symmetry and transitivity of equality, together with function and predicate substitutivity) and the Peano axioms:

$$\underset{\textit{is-nat}[x]}{\forall} \bigwedge \begin{cases} \textit{is-nat}[0] & (\textit{generation:zero}) \\ \textit{is-nat}[x^{+}] & (\textit{geneneration:successor}) \end{cases},$$

$$\underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \begin{cases} x^{+} \neq 0 & (\textit{unique zero}) \\ (x^{+} = y^{+}) \Leftrightarrow (x = y) & (\textit{unique successor}) \end{cases},$$

$$\underset{\mathfrak{F}}{\forall}((\mathfrak{F}[0] \wedge \underset{\textit{is-nat}[x]}{\forall}(\mathfrak{F}[x] \Rightarrow \mathfrak{F}[x^{+}]) \Rightarrow \underset{\textit{is-nat}[x]}{\forall}\mathfrak{F}[x]). \qquad (\textit{induction principle})$$

  In the above, (*induction principle*), is an *axiom* that can not be expressed in first order predicate logic, predicate $\mathfrak{F}$ is a higher order variable. Here, $\mathfrak{F}[x]$ means that x occurs free in the formula denoted by $\mathfrak{F}$. As mentioned, in such situations the common practice is to *lift these axiom schemes to the level of inference rules*, i.e. add the corresponding inference rule to the inference engine, and eliminate the induction axiom from the knowledge base.
- The inference engine for natural numbers $\mathcal{IR}_{\mathbb{N}}$ will consist of the predicate logic inference rules (a natural deduction-style calculus), together with rewriting (to handle equalities), and the induction inference rule, which can be easily reformulated from (*induction principle*):
  - To prove, for any property (predicate) $\mathfrak{F}$, $\underset{\textit{is-nat}[x]}{\forall}\mathfrak{F}[x]$ from the knowledge base $\mathcal{KB}$,
$$\mathcal{KB} \vdash \underset{\textit{is-nat}[x]}{\forall}\mathfrak{F}[x],$$
  - first prove the property holds for 0, under the same knowledge base,
$$\mathcal{KB} \vdash \mathfrak{F}_{x \leftarrow 0},$$
  - then assume the property holds for an arbitrary but fixed new constant x_0, and show it also holds for ${x_0}^{+}$,
$$\mathcal{KB}, \mathfrak{F}_{x \leftarrow x_0} \vdash \mathfrak{F}_{x \leftarrow {x_0}^{+}}.$$
    In the above, $\mathfrak{F}_{x \leftarrow t}$ is the formula obtained by substituting the term t for the variable x, in $\mathfrak{F}[x]$.

Algorithm schemes that reflect the inductive structure of the domain can now be constructed. They are inductive definitions of function symbols, with arity 1, 2, etc. For schemes corresponding to functions with arity greater than 2, any of the arguments can be the inductive one.

For natural numbers we already presented two such schemes in Example 3:
is-rec-nat-binary-fct-1r and *is-rec-nat-binary-fct-1l* are algorithm schemes for binary function symbols in terms of unary auxiliary symbols. The following is a scheme for a binary function symbol expressed in term of unary and binary auxiliary function symbols:

$$\underset{f,g,h}{\forall}(\textit{is-rec-nat-binary-fct-12r}[f,g,h] \Leftrightarrow \underset{\textit{is-nat}[x,y]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x,0] = g[x] \\ f[x,y^+] = h[x,y,f[x,y]] \end{array} \right.),$$

and similar schemes can be added to the library analogously. In fact, since the generation process is clear, these schemes need not be stored in the library, but can, in principle, be generated on demand.

Using Buchberger's scheme based exploration model, outlined in Section 2.1, as shown in [Hodorog and Crăciun, 2007], new well-founded (partial) orders can be added to the theory, e.g. $<$ (smaller than), $|$ (divides), thus allowing the addition of new recursive definition structures - and new (well-founded) induction inference rules. With this, a broad range of new schemes can be added to the library of schemes, e.g.:

$$\underset{f,g,h}{\forall}(\textit{is–nat–step–recl–fct–1–1}[f,g,h] \Leftrightarrow \underset{\substack{\textit{is–nat}[x,y] \\ y>0}}{\forall}(f[x,y] = \left\{ \begin{array}{ll} g[x] & \Leftarrow x < y \\ h[f[x-y,y]] & \Leftarrow \text{otherwise} \end{array} \right.)),$$

which is a recursive algorithm scheme based on the well-founded ordering $<$. This scheme is used to introduce the quotient and remainder functions for natural numbers, in [Hodorog and Crăciun, 2007].

The idea of this scheme is that in order to compute the binary function f, in the special case the first argument is smaller (w.r.t. $<$) than the second, compute '$g[x]$', otherwise apply recursively f on a smaller argument (and we know from the properties of '$-$' that for any x, y, with $x \not< y$, $x - y < y$).

To summarize, this example shows how libraries of algorithm schemes can be built-up systematically by analysis of the structure of inductive domains. Such libraries grow dynamically with the growth of the theory. Indeed, scheme libraries are used to introduce notions in theories (and solve problems), which can add structure to the domain which in turn can add new schemes to the libraries.

Example 6 (Divide and Conquer)**.** One of the basic programming idea is *divide and conquer*: To solve a problem, if the input is simple, then do it in a simple way, otherwise, split the problem into simpler subproblems, solve them and then compose the result to

get your answer. Some conditions have to hold, e.g. the fact that the subproblems are smaller than the initial problem (w.r.t. some well-founded ordering), that it is known what a simple imput is (in the respective domain).

The divide and conquer idea, in its most abstract form, can be expressed as a scheme in the following way:

$$\underset{\substack{id,od,\prec \\ is\text{-}well\text{-}founded[\prec,id]}}{\forall} \underset{f,sp,ls,rs,c}{\forall} (is\text{-}divide\text{-}and\text{-}conquer[id,od,\prec][f,sp,ls,rs,c] \Leftrightarrow$$

$$\underset{id[x],od[y,z]}{\forall} \bigwedge \left\{ \begin{array}{l} f[x] = \left\{ \begin{array}{lcl} sp[x] & \Leftarrow & is\text{-}minimal[x,\prec] \\ c[f[ls[x]],f[rs[x]]] & \Leftarrow & \text{otherwise} \end{array} \right. \\ \\ \begin{array}{c} od[sp[x]] \\ id[ls[x]] \\ id[rs[x]] \\ od[c[y,z]] \\ ls[x] \prec x \\ rs[x] \prec x \end{array} \end{array} \right.),$$

i.e. the divide and conquer scheme:

- defines a function f with arguments from a domain described by the unary predicate id, with the result described by the unary predicate od,
- such that on the domain id we have $\prec$, a well founded ordering on id, described by the higher order binary predicate *is-well-founded*,
- in terms of the functions (subalgorithms) sp, used to calculate the result for minimal elements w.r.t. the well founded ordering ($is\text{-}minimal[x,\prec]$),
- and ls, rs - the function that split the input, and c - the function that combines the partial results, for nonminimal elements,
- such that certain "type" (closure, signature) properties hold for the subalgorithms,
- and also the recursive calls are made on smaller objects (w.r.t. $\prec$).

The instantiation of this scheme with a well-founded ordering and unary predicates that characterize input output domains, gives the divide-and-conquer scheme for the respective domains.

The structure of the input variable can be arbitrarily complicated, e.g. we we could have a binary, ternary, etc. function. Then the well founded ordering has to be defined on pairs of 2, 3, etc. objects. Also, the number of splitting functions could be different, one could have one, two, etc. In fact it can easily be checked that the '*is-nat-step-recl-fct-1-1*' scheme in Example 5, with '$<$' as the well-founded ordering on natural numbers, and '$-$' as a splitting function is an instance of divide-and-conquer.

In fact, continuing the systematic build-up of algorithm schemes, together with their corresponding theories, one can arrive at divide-and-conquer schemes for the respective theories. This shows that while divide and conquer is one of the best known programming

techniques/ideas, a systematic bottom-up analysis and build-up of domain structure can discover it.

The structure of *algorithmic theories*, i.e. theories where computation can be performed, can help with the systematic build-up of algorithm schemes, as shown in the previous example. But can all algorithmic ideas be synthesized in this way?

In principle, the systematic build-up can be continued, by adding more structure to the schemes, etc. However, this author feels that this will lead to an explosion in the number of schemes, without much benefit for the problem solving aspect. Humans still play an essential role in the invention of algorithmic ideas.

Example 7 (Critical-Pair/Completion)**.** The *critical-pair / completion* (CPC) algorithmic idea can be applied in the following setting (as described in [Buchberger, 1987]):

- a domain of objects (*"patterns"*),
- together with a *"reduction"* relation $\rightarrow_F$, generated by a set F of finitely many patterns,
- and a *"word problem"* on the respective domain, i.e. for any objects s, t, is (s, t) in the reflexive, symmetric, transitive closure of the reduction relation $(\leftrightarrow^\star{}_F)$?

The CPC idea is:

- construct a set G such that $\leftrightarrow^\star{}_F \equiv \leftrightarrow^\star{}_G$ and $\rightarrow_G$ has the Church-Rosser property (for these relations the word problem can be decided),
- by starting from F and a set of *"critical situations"*, for which the Church-Rosser property is checked,
- if the property holds, the next critical situation is checked,
- otherwise, the set is *"completed"* with an element that makes the property hold.

This ideea was applied independently to solve problems in automated theorem proving (resolution [Robinson, 1965]), polynomial ideal theory (Gröbner bases), and word problems in universal algebras (the Knuth-Bendix procedure [Knuth and Bendix, 1970]). An overview of the common features of CPC algorithms is given in [Buchberger, 1987].

Let us consider, in the case of polynomial ideal theory, the CPC scheme. In the following, uppercase letters represent polynomial sets, lowercase letters represent polynomials. Since the domain is chosen, the CPC scheme can be formulated if the theory already contains certain concepts:

- a *well-founded ordering on polynomials*, $\prec$,
- a *reduction relation modulo a polynomial set* F, $\rightarrow_F$, such that if $f \rightarrow_F g$, then $g \prec f$,
- a *reduction operation* modulo a polynomial, rd, such that for $g \in F$, $f \rightarrow_F rd[f, g]$,
- a *total reduction operation* modulo a polynomial set, trd, such that $f \rightarrow^\star_F trd[f, F]$, and $trd[f, F]$ is not reducible,
- a *pairing function pairs*, that applied to a polynomial set F gives the set of pairs formed with distinct elements,

- $\frown$ to add to a set of polynomials a new polynomial,
- and *cmp*, to add to a set of pairs of polynomials a set of all pairs obtained from an element and a set of polynomials.

The CPC scheme for polynomial ideals, similar to that given in [Buchberger, 2004b], is:

$$\begin{array}{l} \underset{A,lc,df}{\forall} \textit{is-CPC-poly-scheme}[A, lc, df] \Leftrightarrow \\ \quad \underset{F,g_1,g_2,\overline{p}}{\forall} \bigwedge \left\{ \begin{array}{l} A[F] = A[F, pairs[F]] \\ A[F, \langle\langle g_1, g_2\rangle, \overline{p}\rangle] = \\ \quad \text{where}[f = lc[g_1, g_2], h_1 = trd[rd[f, g_1], F], h_2 = trd[rd[f, g_2], F], \\ \quad \left\{ \begin{array}{lll} A[F, \langle\overline{p}\rangle] & \Leftarrow & h_1 = h_2 \\ A[F \frown df[h_1, h_2], cmp[\langle\overline{p}\rangle, df[h_1, h_2], F]] & \Leftarrow & \text{otherwise} \end{array} \right. \end{array} \right. \end{array}.$$

In addition, the following is required for the *lc* subalgorithm:

$$\underset{g_1,g_2}{\forall} \bigwedge \left\{ \begin{array}{l} rd[lc[g_1, g_2], g_1] \prec lc[g_1, g_2] \\ rd[lc[g_1, g_2], g_2] \prec lc[g_1, g_2] \end{array} \right. .$$

Reasoning about Algorithm Schemes

Using algorithm schemes amounts to proposing solutions to problems. Proving that a proposal solves a problem involves reasoning about the instantiation of an algorithm scheme.

Some sort of induction will be used when reasoning about recursive schemes, e.g.:

- structural induction on natural numbers for *is-rec-nat-binary-fct-12r*,
- course of value induction on natural numbers (with the smaller than relation) for *is–nat–step–recl–fct–1–1*,
- well founded induction (with respect to a well founded ordering on an appropriate domain) for *is-divide-and-conquer*.

Since any such induction is, in fact, an instance of a well-founded induction rule, the selection of the appropriate one to use when reasoning about the scheme amounts to finding the appropriate well-founded ordering on the terms on which the recursive call is used.

2.2.3 The Lazy Thinking Method (Cascade)

Lazy thinking is a method for exploration of synthesis situations (language, knowledge base, inference rules, libraries of algorithm schemes). A lazy thinking exploration step is carried out by the application of:

The Lazy Thinking Cascade

- *Start* with a *synthesis situation*, i.e.:
  - a problem (specification) $\underset{d[x]}{\forall} P[x, A[x]]$ (A is a new symbol, the desired algorithmic solution of P),
  - the theory $\mathcal{T} = \left\langle \mathcal{L}, \mathcal{KB}, \mathcal{IR} \right\rangle$ that corresponds to the problem (the problem is well-understood, i.e. all relevant concepts and properties are in $\mathcal{T}$),
  - a library of algorithm schemes.
- *Select one of the algorithm schemes* from the library of schemes, instantiate it with A, and new symbols for the subalgorithms, and add this instantiation to the knowledge base.
- *Set up the correctness proof*, i.e. a proof of $\underset{d[x]}{\forall} P[x, A[x]]$, using the knowledge available. The proof method to be applied is suggested by the selected algorithm scheme. Note that the proof is likely to fail, due to the fact that we reason about concepts about which we have no knowledge (the subalgorithms from the algorithm scheme).
- **While** the proof fails **do**:
  - *Analyze the failed proof*, and
  - *Generate a conjecture* from the failure, that will allow the proof to get over the failing situation. Add the conjecture to the knowledge base and try the proof again.
- When the proof is completed, *the result of the lazy thinking exploration* round is:
  - a proof of the correctness theorem (i.e. the instantiation of the selected scheme solves the problem) provided that
  - notions that satisfy the list of generated conjectures (if these contain symbols for the unknown algorithms) can be retrieved from the knowledge base or can be invented (synthesized), and the conjectures that do not contain unknown symbols can be proved.

2.2.4 Proof Failure Analysis

In order to explain how proof analysis is done, we will briefly discuss what a proof is (for the purpose of applying lazy thinking). Then we will discuss when does a proof fail, how do we analyze its failure, and what the result of this analysis is.

Proofs

There are several ways to formally define what a proof is, depending on the proof theoretic context, the definition of inference rule, etc., see [Buss, 1998] for an introduction. Here we do not choose a particular formulation of what a proof is. Rather we outline some requirements that a proof should fulfill, and point out some formalisms that

conform.

A *proof* consists of the application of a finite number of reasoning steps, that transform an *initial proof situation*, described by a knowledge base and a goal (or more), into a *final proof situation*(s), where the goal is trivial (true). Note that we consider here Gentzen-style proofs (following the terminology of [Buchberger, 1991]), i.e. proofs that are formed by subsequent transformations of proof situations.

Sequent calculus and natural deduction are examples of such proof formalisms, as well as the "usual" human mathematical proofs, provided that they observe some degree of formalism, i.e. the way the proof is formed can be described as the application of a well-established set of simple inference rules.

Remember that the language context of proving is that of predicate logic, so this set of rules will include predicate logic inference rules, equality proving and domain dependent inference rules.

A *proof object*, i.e. what results from an attempt to prove, can best be represented as a (deduction) tree where nodes are proof situations and (directed) edges indicate the application of an inference rule. Given a proof situation, several inference rules can be applied, which results into an OR node (the respective proof situation is "proved" if either of the subtrees lead to a trivial proof situation — i.e. the goal appears in the knowledge). In other situations, the application of an inference rule may yield several new proof situation – an AND node – where the given proof situation is "proved" if all the subtrees are "proved" (lead to trivial proof situations). For the formal details of deduction trees, see [Tomuta, 1998].

Proof failures and their analysis

A proof *fails* when a nontrivial proof situation cannot be transformed by any of the available inference rules. This means that neither the goal, nor the knowledge can be changed.

In the lazy thinking method, which we consider here, the idea is to "force" the proof to get over the failure. This means that some extra knowledge should be added to the knowledge base, such that the failure is overcome.

The strategy we use to generate this knowledge is the following:

- in the course of generating the proof, *collect temporary knowledge* generated, $\mathcal{TKB}$ - this will be a set of formulae,
- get the failing goal: $\mathcal{G}$ - this will be one formula,
- construct the following *conjecture skeleton*: $\{\mathcal{G}, \mathcal{TKB}\}$. The conjecture that will allow the proof to continue will be generated from this skeleton.

Not all (temporary) knowledge generated in the course of the proof will be considered:

- Those formulae in the knowledge which are existentially quantified can be easily

transformed in ground formulae (with arbitrary but fixed constants),
- Those formulae that are universally quantified implications or equivalences can be used as generalized rewriting rules (e.g. modus ponens, modus tollens, or the PC method, proposed by Bruno Buchberger, see [Buchberger et al., 2000]), to produce new knowledge. Therefore such formulae will not be considered in the temporary knowledge. However, the knowledge produced using these rules will be considered (after the elimination of quantifiers).
- In addition, universally quantified formulae that cannot be used for knowledge rewriting can be used as temporary knowledge (see below, Subsection 2.2.5 for how this can be used in the generation of the conjectures).
- The body of temporary knowledge can further be pruned by keeping only those formulae that contain notions of interest — for instance about unknown subalgorithms from an algorithm scheme).

A proof object is a tree, and our proposed strategy analyzes only one branch of this tree — the one leading to the (first) failure. In case the proof tree has several branches, we choose the **first failed branch**.

If branching was due to an AND node, then from the first failed branch a conjecture will be generated that will allow the respective branch to succeed. The "open" ("pending") branches will then be reached and dealt with in the same manner.

If, however, branching was due to an OR node, then several inference rules were applicable. The first failed branch approach will yield one conjecture, which depends on which of the inference rules was applied first (i.e. the proof strategy). In this case, a bad proof strategy may lead to a bad (e.g. useless) conjecture. Note that this is no worse than the general situation in proving: a bad choice of strategy can lead to a bad result.

To summarize, the proof analysis method works in the context of Gentzen-type natural style proof mechanisms, in the language frame of predicate logic with equality.

The result of the analysis of a failed proof situation is a conjecture skeleton: we use a first failed branch strategy to select the situations to analyze. In this way we avoid an explosion of the exploration space, and the case studies we have carried out(see Example sections in this chapter, also Chapter 4, and [Buchberger and Crăciun, 2004a], [Buchberger and Crăciun, 2004b], [Hodorog and Crăciun, 2007]) have shown that our strategy is sufficiently powerful to achieve interesting results.

The conjecture skeleton is essentially a set of formulae, temporary assumptions (ground and/or universally quantified), and the failing goal (ground formula). The set of temporary assumptions is chosen according to filters described above. Several of these filters can be applied.

2.2.5 Conjecture Generation

The conjecture generation phase uses the output of the failure analysis phase, i.e. the conjecture skeletons $\{\mathcal{G}, \mathcal{TKB}\}$, with $\mathcal{G}$ a formula representing a failing goal, and $\mathcal{TKB}$ a list of temporary assumptions made during the failing proof.

From these, using *generalization strategies*, generalization substitutions are assembled, which will be applied to the skeletons, from which conjectures will be generated (usually an implication with the left hand side obtained by generalizing the temporary knowledge $\mathcal{TKB}$ and the right hand side obtained from generalizing the goal $\mathcal{G}$).

The aim is to generate such a conjecture that will allow the proof to get over the failure. Intuitively, since the conjecture is formed from the implication mentioned above, when it is added to the knowledge base and the proof is attempted again, the generalization of the temporary knowledge can be specialized again, and so can the generalization of the goal. Thus, the proof will get over the failure point.

As a general principle, when generating a conjecture, this must be such that a (series of) reasoning step(s) which involves this conjecture can be applied to get over the initial failure. Therefore, several generalization strategies are possible. We present some below.

We distinguish between two situations, according to the shape of the result of failure analysis:

- ground skeletons (i.e. no variables occur in the formulae of the skeletons) and
- skeletons with universally quantified formulae in the temporary knowledge.

Generalization strategies also use information found in algorithm schemes, e.g. the list of unknown symbols (denoting the subalgorithms), the new symbol (the algorithm being synthesized).

Ground Skeletons

The first generalization strategy (*term generalization*) will be applied in the case of *recursive algorithm schemes*. This consists of the following:

- identify in the conjecture skeleton's (goal) all the terms of the form

$$n[\ldots, aux[\ldots], \ldots],$$

- generate the substitution

$$n[\ldots, aux[\ldots], \ldots] \rightarrow x,$$

  where n is the new symbol being defined by the algorithm scheme, aux is one of the subalgorithms from the algorithm scheme, x is a new variable which is of the same type as the term.

The intuition behind this generalization is the following: By generalization we want to say something about the term. Since we apply the unknown new algorithm to a term which is the result of the application of one of the unknown subalgorithms, all we can say about this result is that it will have the same output type as the algorithm being defined. Therefore, we generalize it to a new variable. The result of this phase is a substitution τ.

The second generalization strategy (*constant generalization*), consists of identifying the arbitrary but fixed constants in the skeleton and generalize them to new variables of the corresponding type. The result of this phase is a substitution σ.

Note that this generalization strategy is very weak, arbitrary but fixed constants come from the elimination of quantifiers, and by applying the generalization substitution we just re-introduce these quantifiers. That's why this generalization strategy should be used in connection to others, like the term generalization described above. Otherwise, the generated conjecture will not be very useful (a "dud").

The generalization substitutions σ resulting from the generalization strategies are applied to the skeleton, and we construct a formula in the following manner:

$$\underset{r}{\forall}(\mathcal{TKB}_{\tau,\sigma} \Rightarrow \mathcal{G}_{\tau,\sigma}),$$

where

- r is the variable range containing all the variables (together with their "type" information) that are obtained from the generalization phase (by application of the substitutions τ, σ),
- $\mathcal{TKB}_{\tau,\sigma}$ is obtained from $\mathcal{TKB}$ by applying first the substitution τ, then the substitution σ, and taking the conjunction of the result (from a list of ground formulae we obtain a conjunction of formulae with variables),
- $\mathcal{G}_{\tau,\sigma}$ is obtained from $\mathcal{G}$, by applying first the substitution τ, then the substitution σ.

Universal Quantification in Temporary Knowledge

If the temporary knowledge part of the skeleton $\{\mathcal{G}, \mathcal{TKB}\}$ has the form

$$\mathcal{TKB} : \ldots, \underset{s}{\forall}F, \ldots,$$

where s is the variable range containing all the variables in formula F, i.e. the temporary knowledge contains an universally quantified formula which cannot be used for knowledge rewriting,

and

$\underset{s}{\forall}F$ matches the (ground) goal formula $\mathcal{G}$, yielding a substitution

$$\varphi = \{v \leftarrow t | v \in s, t \text{ a term occuring in } \mathcal{G}\}$$

then

the generalization strategy is the following: delete the matching formula $\underset{s}{\forall}F$ from $\mathcal{TKB}$ (which now becomes $\mathcal{TKB}'$) and delete $\mathcal{G}$, replacing it with $\underset{s}{\exists}F'$, where the F' is obtained from φ by transforming each substitution $v \leftarrow t$ from φ into the equality $v = t$, and taking their conjunction.

Following this step, apply the term generalization and variable generalization strategies described above (obtaining the substitutions τ, σ as described above). Then the conjecture is given by:

$$\underset{r}{\forall}(\mathcal{TKB}'_{\tau,\sigma} \Rightarrow \underset{s}{\exists}F'_{\tau,\sigma}),$$

where:

- r is the variable range containing all the variables (together with their "type" information) that are obtained from the generalization phase (by application of the substitutions τ, σ),
- $\mathcal{TKB}_{\tau,\sigma}$ is obtained from $\mathcal{TKB}$ by applying first the substitution τ, then the substitution σ, and taking the conjunction of the result (from a list of ground formulae we obtain a conjunction of formulae with variables).
- $\underset{s}{\exists}F'_{\tau,\sigma}$ is the formula obtained by the application of the substitution τ, then the substitution σ to $\underset{s}{\exists}F'$.

The conjecture generated this way will get over the failure:

- the generalization of the temporary knowledge will be again specialized, and by modus ponens $\underset{s}{\exists}F'$ will be added to the knowledge at a certain point in the proof,
- $\underset{s}{\exists}F'$, after the elimination of the existential quantifier, will provide equalities that will allow the instantiation of $\underset{s}{\forall}F$, and its subsequent rewriting into $\mathcal{G}$, thus completing the proof. The fact that the rewriting process will lead to the goal is guaranteed by the fact that $\underset{s}{\forall}F$ and $\mathcal{G}$ match.

Note that this generalization strategy is similar with the *semantical pattern matching* used in the frame of Bruno Buchberger's PCS method, see [Buchberger et al., 2000].

Generalization Strategies

The generalization strategies presented above are just some of potentially many others. These strategies were devised during our case studies, such as those presented in the following sections, as well as in Chapter 4.

How (Many) To Generate/Choose?

As discussed in the previous Subsection 2.2.4, at the failure analysis stage a first failed strategy is chosen. This may lead (in the case the proof object branches on an OR node) to the choice of a bad (useless) skeleton, leading to a bad (useless) conjecture. One way to address this is to generate several skeletons (corresponding to all proof branches), and see which leads to better results.

However, since a bad conjecture would be a result of a bad (i.e. unsuitable for the problem at hand) proof strategy, generating several skeletons amounts to trying out several proof strategies. In other words, the first failure strategy (our choice) is sufficient for generating conjectures, and fine tuning of the conjecture generation machinery can be done via the proof strategies.

2.2.6 Using Lazy Thinking

We now give some considerations on how to use the lazy thinking method.

Regarding the inferences, or proof strategies, this method will work for Gentzen-style proof system. Moreover, given the way our conjectures are generated from failing proofs, a proof strategy that makes use of "generalized" (or knowledge) rewriting, i.e. rewriting using equivalences, implications, equalities, such as described in Buchberger's PCS method, the C phase, see [Buchberger, 2001].

Figure 2.1 presents the way in which lazy thinking can be used. Boxes represent exploration situations, elipses represent explorations (actions). Note that the algorithm schemes — although an important part of the method — are not compulsory:

(a) Using algorithm schemes directly: the instantiation of the algorithm scheme is added to the knowledge (as described above in this Section).
(b) No explicit scheme instantiation is added to the knowledge. Rather, in a preprocessing (exploration) step, derive some properties of the scheme, and add these to the theory. In our main case study, in Chapter 4 we illustrate such a situation.
(c) No scheme instantiation is added to the knowledge - thus providing no information on the desired solution. In general this will complicate finding the solution for the problem, and the term generalization strategy will not really work (no information on auxiliary symbols means no terms are generalized, the rule is not applicable). While we do not exclude such use, we did not pursue this line of investigation, which gives up the main advantage of schemes (explicit algorithmic knowledge from algorithm schemes), and relies on implicit knowledge. As discussed in Chapter 5 (5.2.5) similar approaches have already been investigated, but although successful, they were re-inventing well-known algorithmic principles.

The lazy thinking method is necessarily not complete, and we only considered a couple of strategies for the essential step of conjecture generalization. In our case studies, we applied the strategies incrementally, depending on the result of the exploration process.

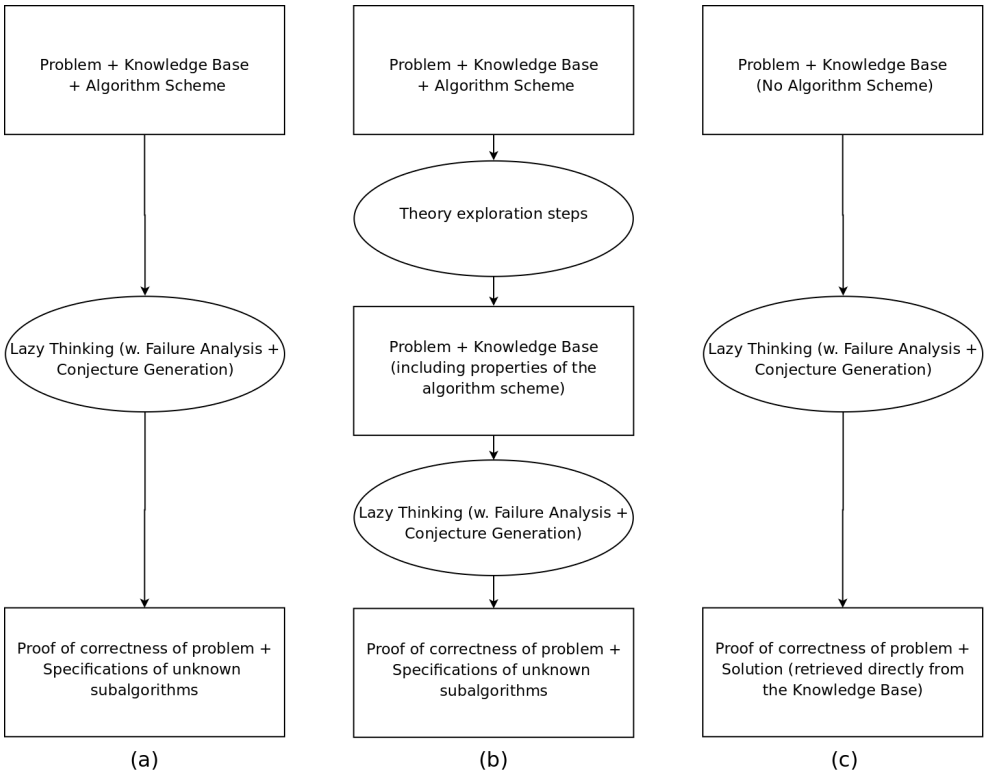

Figure 2.1: Using lazy thinking.

Lazy thinking can fail if no new conjectures are generated (i.e. either the conjecture was generated before, or we get essentially the same problem).

Lazy thinking can also detect cases when no solution for the problem exists: Because of a contradiction, the proof of the correctness statement fails.

For the rest of this chapter, we provide some examples where we apply lazy thinking, illustrating the various ways to use the method.

2.3 Example: Synthesis of an Algorithm

In our first example, we illustrate how to use lazy thinking to solve problems (synthesize algorithmic solutions) where the proposed algorithmic solution is based on the "divide and conquer" idea. By leaving the problem abstract (we only specify some conditions that have to hold), we obtain generic solutions for all problems that can be solved by "divide and conquer". We then indicate how these generic solutions (specifications) can be used in the synthesis of concrete problems.

2.3.1 Synthesis Situation

Consider two arbitrary domains described by the predicates I, O, respectively, and $\prec$ such that *is-well-founded*$[\prec, I]$, a well-founded partial ordering on the first domain.

Let P be a predicate. We want to solve the following **problem**:

$$\underset{I[X]}{\forall} P[X, F[X]], \qquad \text{(P)}$$

i.e. we want to find an algorithm F, whose output domain is described by the predicate O. We call formula (P) the correctness theorem or problem specification.

Now, consider the *is-divide-and-conquer* scheme from Subsection 2.2.2, available as part of the library of algorithm schemes.

Since I, O, $\prec$ satisfy the conditions in the formulation of the scheme, we propose as a solution to our problem (P) the instantiation:

$$\textit{is-divide-and-conquer}[I, O, \prec][F, s, h1, h2, g],$$

where s, $h1$, $h2$, g are new symbols in the language of the theory, representing the subalgorithms in the definition of the algorithm.

This intantiation brings the following **knowledge** (denoted by $\mathcal{KB}$) in the theory:

- the proposed definition for the algorithm:

$$\underset{I[X]}{\forall} F[X] = \begin{cases} s[X] & \Leftarrow \textit{is-minimal}[X, \prec] \\ g[F[h1[X]], F[h2[X]]] & \Leftarrow \text{otherwise} \end{cases}, \qquad \text{(DC)}$$

- the input-output conditions for the subalgorithms:

$$\underset{\substack{I[X] \\ \textit{is-minimal}[X,\prec]}}{\forall} O[s[X]], \qquad \text{(S.tr.dom)}$$

$$\underset{\substack{I[X] \\ \neg\textit{is-minimal}[X,\prec]}}{\forall} \bigwedge \begin{cases} I[h1[X]] \\ I[h2[X]] \end{cases}, \qquad \text{(Spl.ntr.dom)}$$

$$\underset{O[X_1,X_2]}{\forall} O[g[X_1, X_2]], \qquad \text{(C.dom)}$$

- the splitting functions $h1$, $h2$ are smaller (w.r.t. $\prec$) than their argument:

$$\underset{\substack{I[X] \\ \neg\textit{is-minimal}[X,\prec]}}{\forall} \bigwedge \begin{cases} h1[X] \prec X \\ h2[X] \prec X \end{cases}. \qquad \text{(Spl.ntr.ord)}$$

The **inference mechanism** we use is well-founded induction w.r.t. $\prec$, together with predicate logic with equality.

Note that the synthesis situation is now completely described.

2.3.2 Lazy Thinking Exploration Rounds

We now set up lazy thinking. We try to prove the correctness statement, when the proof fails, analyze the failure and generate a conjecture that allows the proof to go on.

First cascade iteration

Proof attempt. Try to prove (P) under the knowledge $\mathcal{KB}$, by well-founded induction w.r.t. $\prec$:

We take X_0 arbitrary but fixed such that $I[X_0]$ and assume as induction hypothesis:

$$\underset{\substack{I[X]\\ X \prec X_0}}{\forall} P[X, F[X]].$$

We try to prove

$$P[X_0, F[X_0]].$$

Case *is-minimal*$[X_0, \prec]$:

In this case, by the definition of F, it suffices to prove:

$$P[X_0, s[X_0]].$$

At this point, the proof is stuck because there is no specific knowledge available on s that could transform the proof situation further. □

Failing proof analysis and conjecture generation.

Applying the failure analysis method described earlier in this chapter proposed we collect the current assumptions and the unproved goal in the conjecture skeleton

$$\{P[X_0, s[X_0]],\ \{I[X_0], \textit{is-minimal}[X_0, \prec]\}\},$$

and conjecture

$$\underset{\substack{I[X]\\ \textit{is-minimal}[X, \prec]}}{\forall} P[X, s[X]], \qquad \text{(Sp.tr)}$$

by applying the constant generalization:

$$X_0 \to X,$$

where 'X' is a new variable. No term generalizations were necessary in this case.

(Sp.tr) is a specification for s which is added to $\mathcal{KB}$, which clearly will make the proof succeed in this situation (we omit the details).

Second cascade iteration

Proof attempt (continued).

Case $\neg$*is-minimal*$[X_0, \prec]$:

By the definition of F, it suffices to prove:

$$P[X_0, g[F[h1[X_0]], F[h2[X_0]]]].$$

From the instantiation of the input-output conditions of F, and $h1$, $h2$ – (Spl.ntr.dom), we know:

$$O[F[h1[X_0]]] \text{ and } O[F[h2[X_0]]].$$

By modus ponens, using (Spl.ntr.dom), (Spl.ntr.ord) and the well–founded induction hypothesis we know:

$$P[h1[X_0], F[h1[X_0]]] \text{ and } P[h2[X_0], F[h2[X_0]]].$$

The proof is stuck, the proof situation cannot be transformed anymore.

□

Failing Proof Analysis and Conjecture Generation.

We analyze the failing proof attempt, collect the current assumptions and the unproved goal in the conjecture skeleton:

$$\{P[X_0, g[F[h1[X_0]], F[h2[X_0]]]], \left\{ \begin{array}{l} I[X_0] \\ \neg \textit{is-minimal}[X_0, \prec] \\ O[F[h1[X_0]]] \\ O[F[h2[X_0]]] \\ P[h1[X_0], F[h1[X_0]]] \\ P[h2[X_0], F[h2[X_0]]] \end{array} \right\}\}$$

and generalize it to:

$$\underset{\substack{I[X], O[X_1, X_2] \\ \neg \textit{is-minimal}[X, \prec]}}{\forall} \left(\bigwedge \left\{ \begin{array}{l} P[h1[X], X_1] \\ P[h2[X], X_2] \end{array} \right. \Rightarrow P[X, g[X_1, X_2]] \right), \qquad \text{(Sp.ntr)}$$

by first applying the term generalizations

$$F[h1[X_0]] \to X_1,$$
$$F[h2[X_0]] \to X_2,$$

and then the constant generalization $X_0 \to X$, where 'X', 'X_1', 'X_2' are new variables.

(Sp.ntr) is added to $\mathcal{KB}$, and in fact it is easy to see that now the proof will succeed (details ommited).

Result of lazy thinking

After a third iteration of the cascade which ends with the proof of the correctness theorem (P) (omitted here), lazy thinking ends with success. We have a proof of the correctness theorem (P), for our proposed solution (DC), *provided that* we can find subalgorithms s, $h1$, $h2$, g that verify the generated specifications (Sp.tr), (Sp.ntr).

2.3.3 Discussion

The example we gave here is significant, in that it considers an abstract setting (problem). The result of the application of lazy thinking can be used in any concrete situation that fits this setting. This is an example of *preprocessing* an algorithm scheme.

In fact we showed in [Buchberger and Crăciun, 2004b] how we can apply this result to the synthesis of sorting algorithms for tuples, but also in subsequent applications of lazy thinking to the invention of the subalgorithms that appear in the proposed definition of the sort function: synthesis of the splitting functions, and that of the merging function (to synthesize the merge-sort algorithm).

2.4 Example: Unsatisfiable Specification

In the second synthesis example, we propose a problem in the theory of natural numbers. We ask whether natural numbers together with the addition and 0 and some unary (unknown) function form a group. It is easily provable that naturals with addition and 0 form a monoid, therefore the problem reduces to checking whether the unknown unary function is an inverse for naturals, with respect to addition. By applying lazy thinking we show that for natural numbers such an inverse does not exist.

2.4.1 Synthesis Situation

Going back to the problem in the theory of natural numbers we presented in Example 4, *is-group*$[$*is-nat*$, +, 0, \ominus]$, this reduces to the **problem**:

$$\underset{\textit{is-nat}[x]}{\forall} x + \ominus x = 0,$$

where $\ominus$ is a new function symbol.

The **knowledge base** $\mathcal{KB}_{\mathbb{N}}$ consists of:

- the axioms for natural numbers (generation and uniqueness):

$$\begin{array}{ll} \textit{is-nat}[0] & \text{(gen. zero)} \\ \underset{\textit{is-nat}[x]}{\forall} \textit{is-nat}[x^+] & \text{(gen. succ)} \\ \underset{\textit{is-nat}[x]}{\forall} x^+ \neq 0 & \text{(uniq. zero)} \\ \underset{\textit{is-nat}[x],\textit{is-nat}[y]}{\forall} (x^+ = y^+) \Leftrightarrow (x = y) & \text{(uniq. succ)} \end{array}$$

- definitions and properties of the function $+$:

$$\begin{array}{ll} \underset{\textit{is-nat}[x],\textit{is-nat}[y]}{\forall} \bigwedge \left\{ \begin{array}{l} x + 0 = x \\ x + y^+ = (x + y)^+ \end{array} \right. , & (def.+) \\ \underset{\textit{is-nat}[x],\textit{is-nat}[y]}{\forall} \bigwedge \left\{ \begin{array}{l} 0 + x = x \\ x^+ + y = (x + y)^+ \end{array} \right. , & (equiv.+) \\ \textit{is-monoid}[\textit{is-nat}, +, 0]. & \end{array}$$

The inference mechanism $\mathcal{IR}_{\mathbb{N}}$ consists of the structural induction for natural numbers and predicate logic with equality inference rules.

For complete details of this exploration, see [Hodorog and Crăciun, 2007].

2.4.2 Lazy Thinking Exploration Rounds

We now begin the lazy thinking exploration. Note that we did not propose any solution for the function $\ominus$. Indeed we could have, since in the theory of natural numbers we have available recursive algorithm schemes. But let's see what the exploration will yield, and if more information will be needed we will provide a proposal for the solution.

First cascade iteration

Proof attempt. Prove

$$\underset{\textit{is-nat}[x]}{\forall} (x \oplus (\ominus x) = 0),$$

using $\mathcal{KB}_{\mathbb{N}}$.

We try to prove by structural induction on x:

Base Case:

Prove $0 \oplus (\ominus 0) = 0$.

By (equiv.+), we have to prove: $\ominus 0 = 0$.

The proof is stuck, no inferences can be applied. □

Failure analysis and conjecture generation

The conjecture skeleton consists of $\{\ominus 0 = 0, \{\}\}$, the goal is an explicit definition which contains no arbitrary but fixed constants, therefore the conjecture is straightforward, $\ominus 0 = 0$, and it trivially allows the proof to go on.

Second cascade iteration

Proof attempt (continued).

Induction Step:

Take x_0 arbitrary but fixed, such that *is-nat*$[x_0]$.

Assume $x_0 + \ominus x_0 = 0$.

Show ${x_0}^+ + \ominus {x_0}^+ = 0$.

By (equiv. $+$) this is equivalent to proving: $(x_0 + \ominus {x_0}^+)^+ = 0$.

However, this is not true, because it contradicts (uniq. zero). □

Result of lazy thinking

The proof attempt shows that the problem we try to solve is contradictory with the theory. Therefore the problem does not have a solution in this theory.

2.5 Example: Invention of a Notion

In the theory of natural numbers we ask whether a certain binary function (multiplication) has a neutral element. We apply lazy thinking to synthesize this element.

2.5.1 Synthesis Situation

We consider a later stage in the development of the theory of natural numbers, which now contains a new binary function symbol ($*$), together with its properties, in the updated **knowledge base** $\mathcal{KB}_{\mathbb{N}}$:

- definitions of the function $*$ (and equivalent forms):

$$\bigwedge_{\substack{\forall \\ \textit{is-nat}[x],\textit{is-nat}[y]}} \begin{cases} x * 0 = 0 \\ x * y^+ = x + x * y \end{cases}, \quad (def.*)$$

$$\bigwedge_{\substack{\forall \\ \textit{is-nat}[x],\textit{is-nat}[y]}} \begin{cases} 0 * x = 0 \\ x^+ * y = x * y + y \end{cases}, \quad (equiv.*)$$

- structural properties introduced by knowledge schemes (and proved):

$$\textit{is-semigroup}[\textit{is-nat}, *].$$

The **inference mechanism** $\mathcal{IR}_{\mathbb{N}}$ is the same as previously: structural induction and predicate logic with equality.

We now propose the following **problem**: $\textit{is-monoid}[\textit{is-nat}, *, c]$, i.e. whether we can find a new constant c with $\textit{is-nat}[c]$ (identity element w.r.t. $*$) such that

$$\underset{\textit{is-nat}[x]}{\forall}(x * c = x).$$

2.5.2 Lazy Thinking Exploration Rounds

We now set up lazy thinking.

First cascade iteration

Proof attempt. Prove

$$\underset{\textit{is-nat}[x]}{\forall}(x * c = x),$$

using $\mathcal{KB}_{\mathbb{N}}$.

We try to prove the goal by structural induction on x:.

Base Case: Prove $0 * c = 0$. This is true by (equiv. *).

Induction step:

Take x_0 arbitrary but fixed such that $\textit{is-nat}[x_0]$.

Induction hypothesis: Assume $x * c = x$.

Induction conclusion: Show $x^+ * c = x^+$.

From (equiv.*), this is equivalent to showing $x * c + c = x^+$.

By the induction hypothesis, and since

$$x^+ \overset{from(equiv.+)}{=} (x + 0)^+ \overset{from(equiv.+)}{=} x + 0^+,$$

we have to show $x + c = x + 0^+$, i.e. $c = 0^+$.

The proof is stuck.

□

Failure analysis and conjecture generation

The conjecture skeleton is $\{c = 0^+, \{\}\}$ which immediately leads to $c = 0^+$. With this, the proof is successful. We invented the new constant $c = 0^+$. We now can give it a more appropriate name (according to taste), e.g. call it 1.

Comments

This small example shows that lazy thinking can invent constants, not only functions. In this case, proposing a solution by an algorithm scheme was not necessary.

CHAPTER 3

Implementation of the Lazy Thinking Synthesis Method in *Theorema*

To implement the lazy thinking method, one needs to implement the main components: the *cascade* that organizes the exploration process, the *proof failure analysis* mechanism, and the *conjecture generation mechanism*. This chapter is dedicated to an implementation of the lazy thinking method in the *Theorema* system.

We start with a short overview of the system, the main section is dedicated to our implementation, and finally we give some remarks on the integration of the lazy thinking method with the available *Theorema* reasoners.

3.1 The Theorema System: Preliminaries

Theorema is a mathematical assistant system that aims at supporting the process of doing mathematics, in all its aspects: proving, solving, computing, exploring mathematical theories, exchanging mathematical information (knowledge bases, publishing), in an integrated language frame, a version of (higher order) predicate logic. Emphasis is put on the natural style interaction and presentation of results.

The system is implemented in *Mathematica* [Wolfram, 2003]: it is written in the *Mathematica* programming language (making use of its powerful pattern matching capabilities), and it uses the advanced *Mathematica* notebook frontend through which the user interacts with the system. However, *Theorema* does *not*, by default, use any of the *Mathematica* algorithms, rather these are used only if explicitly required by the user.

A user formulates mathematical facts in the *Theorema language*, a version of (higher order) predicate logic. *Theorema* provides a two-dimensional notation for this language

that is visually very close to textbook mathematics. Furthermore, *Theorema* contains language constructs, such as **Definition**, **Proposition**, **Theorem**, as keywords for formulae, it also allows to declare free variables (**any**), and to formulate conditions on free variables (**with**).

The *Theorema user language* commands are used to perform *actions* (proving, solving, computing):

$$\textbf{Action}[\textit{entity}, \textbf{using} \rightarrow \textit{knowledge-base}, \textbf{by} \rightarrow \textit{reasoner}, \textit{options}]$$

where **Action** is one of **Prove**, **Solve**, **Compute**. We now focus on proving.

The *Theorema* reasoning tools are a collection of reasoning modules (*Theorema Basic Provers*), i.e. collections of *reasoning rules*:

- *general purpose modules*: such as basic provers for propositional and predicate logic, rewriting,
- *special provers*: they implement reasoning rules specific to certain domains: set theory provers, induction provers, geometry provers, reasoning in real closed fields (Collins' CAD Method).

These reasoning modules can be combined through the *Theorema User Provers* mechanism, which essentially implement proof strategies. These are available to the user in the **Prove** command.

Reasoning rules describe the transformation of *reasoning situations*, represented by a *goal* and *knowledge base* (similar to sequent calculus).

The result of the *Theorema* **Prove** command is a *proof object*, a tree of proof situations, whose root is the initial proof situation (goal and knowledge), of the kind discussed in Chapter 2(2.2.4), see also [Tomuta, 1998]. The proof object is generated by a proof search mechanism that applies the sets of rules from the reasoning modules used in the user prover, according to the combination strategy described in it.

For a comprehensive overview of the *Theorema* system, including technical descriptions of the language, reasoning mechanisms, reasoning object, and description of the available reasoning modules, see the overview papers [Buchberger et al., 1997], [Buchberger et al., 2000], [Buchberger et al., 2006].

For the rest of this work, we will include *Theorema* code and examples from *Theorema* sessions. This Chapter will include code from our implementation of lazy thinking, whereas in Chapter 4 we present a case study, as carried out in *Theorema*.

3.2 Design Principles for This Implementation

Smooth integration into *Theorema*: The implementation of lazy thinking should not introduce a new command in the *Theorema* user language, or change the way the

user interacts with the system. Since lazy thinking is a deductive synthesis method, the most natural way to call it is as a reasoner for a **Prove** command.

Allow "upwards" integration: The implementation of lazy thinking is done in such a way that, in the future, it may be easily integrated into a general framework for scheme based reasoning.

Provide facilities for advanced users/debugging: Lazy thinking is not an always terminating decision method, it may "diverge". The user plays an important part in the exploration process, and our goal is to provide the user with relevant information at any time. Although the lazy thinking exploration is automated, the user should be able to access intermediate results.

The implementation of lazy thinking is organized in the following *Theorema* packages, each corresponding to one component of lazy thinking:

- **CascadeLT**, implements the exploration loop;
- **FailureAnalyzer**, for the analysis of proof attempts;
- **ConjectureGenerator**, for generating conjectures from failing proofs.

In the following, for each package/function that is part of our implementation, we provide:

- **a description for the user**, i.e. what calls are available in a *Theorema* session, and which are the typical situations in which these are used. Some of these calls are intended for "normal" users (and usage), whereas others are for "advanced" users, or for debugging purposes (the intended use will be indicated);
- **an informal description of the package / function**: in particular, for each function, the input/output, the relation with other functions and packages (uses/used by), side effects, and a synopsis are provided. This should give the interested reader an overview of the implementation.
- **the *Theorema* code**, of interest for the system's developers.

Following the detailed presentation of all components of our implementation, we give an overview of the interaction of the various functions in Figure 3.1, included at the end of this chapter.

3.3 The Cascade: CascadeLT

3.3.1 Description for the User

In order to start a lazy thinking exploration process, the user of our implementation in *Theorema*:

- specifies the *problem* P (correctness statement),
- provides the *relevant knowledge base* Kb, including an instantiation of the *algo-*

rithm scheme proposed for solving the problem,
- indicates the *prover* Pr to be used in this process.

Automated lazy thinking exploration is then set up by:

Prove[P, ***using*** → ***Kb***,
by → ***CascadeLT***[Pr, ***GenerateConjectures***, ***nS***, ***auxNS***, ***kConj***, ***nConj***],
ProverOptions → {...}]

where:

- ***GenerateConjectures*** is the function that implements failure analysis and conjecture generation,
- ***nS*** contains the new symbol introduced in the language by the instantiation of the algorithm scheme used (this argument can be empty),
- ***auxNS*** contains the (possibly empty) list of auxiliary new symbols (for subalgorithms) introduced by the instantiation of the algorithm scheme,
- ***kConj*** is a list of conjectures that contain only symbols already in the language, and ***nConj*** is a list of conjectures involving the new symbols. These arguments are present for future "upward" integration of the lazy thinking implementation into a framework for theory exploration. A theory can be validated only when all the conjectures from ***kConj*** have been proved. A synthesis process is complete only when all conjectures from ***nConj*** (which are specifications for the auxiliary symbols) have been solved (i.e. notions were discovered by synthesis or retrieval). At the beginning of algorithm synthesis exploration by lazy thinking these lists empty. They are updated in the recursive calls of the method.

The following commands load the lazy thinking implementation in a *Theorema* session:

```
Needs["Theorema`Provers`Cascade`CascadeLT`"],
Needs["Theorema`Provers`ConjectureGenerator`FailureAnalyzer`"],
Needs["Theorema`Provers`ConjectureGenerator`ConjectureGenerator`"].
```

3.3.2 Implementation: Informal

The package **Theorema`Provers`Cascade`CascadeLT** contains only one function, which implements the automated exploration by lazy thinking,

CascadeLT

INPUT:

This is a curried function, with the first set of arguments:
- the ***Prover*** used for the proof attempt,
- the ***ConjectureGenerator*** for analyzing the proof attempts and generating conjectures,
- ***nS*** the list of new symbols,

- ***auxNS*** the list of auxiliary new symbols,
- ***kConjectures*** the list of conjectures about known symbols,
- ***nConjectures*** the list of conjectures about new symbols,

and the second set of arguments describing the proof situation:

- ***g***, the goal, which is a *Theorema* labelled formula (•***lf***),
- ***kb***, the knowledge base, an assumption list (•***asml***),

OUTPUT:

The (machine processable) output of the cascade is:

$$\{status, kb, nS, auxNS, kConjectures, nConjectures\}$$

where

- ***status*** is one of **"proved"**, **"failed"**, depending on the success of the exploration,
- $\{kb, nS, auxNS, kConjectures, nConjectures\}$ are the updated versions of the corresponding inputs.

USES:

- *Theorema* packages for proof management and formula manipulation:
  - **HandleFormulae**,
  - **UserLanguage**,
  - **ProofObjectInterface**,
- lazy thinking packages
  - **ConjectureGenerator**,
  - **FailureAnalyzer** (indirectly through **ConjectureGenerator**).

USED BY:

- the user through the ***Prove*** command,
- **CascadeLT** is a "top" package, no package calls it directly.

SIDE EFFECTS and COMMENTS:

The integration of **CascadeLT** into the proof command required a modification in the **UserLanguage** package, see below. No other packages were modified.

SYNOPSIS:

- In the initialization phase, prepare the lists of conjectures for processing (knowledge bases need to be flattened).
- Then start an exploration round by calling ***Prover***$[g, kb, ...]$.
- **If** the proof value returned is "proved", success, print a message to the user, display the proof, and return **{"proved", ...}**,
- **Else** begin the analysis of the proof attempt, by calling ***ConjectureGenerator***:
  - **If** the conjecture generator is not successful in generating a conjecture (i.e. it returns **"nothing"** or the generated conjecture is not new), then lazy thinking fails, inform the user, return **{"failed", ...}**,
  - **Else**, add the new conjectures to the knowledge base, update the lists of conjectures, and call the recursively the cascade.

3.3.3 Implementation: Code

Cascade - Lazy Thinking

Package Description

This package contains the implementation of the lazy thinking cascade, which organizes an exploration round using lazy thinking.

```
Needs["Theorema`"]

NewPackage["Theorema`Provers`Cascade`CascadeLT`",
    {"Theorema`Language`General`HandleFormulae`",
    "Theorema`Provers`Common`HandleFormulae`",
    "Theorema`Language`Semantics`UserLanguage`",
    "Theorema`Provers`Common`ProofObjectInterface`",
    "Theorema`System`Debug`",
    "Theorema`Provers`ConjectureGenerator`FailureAnalyser`",
    "Theorema`Provers`ConjectureGenerator`ConjectureGenerator`"}];

ProtectedSymbols[$Context] = Hold[CascadeLT];

ExportedSymbols[$Context] = Join[Hold[], ProtectedSymbols[$Context]];
```

Begin

```
Begin["`Private`"];
```

Supported Properties

The CascadeLT "prover" inherits the properties of the Prover it calls:

```
SupportedProperties[CascadeLT[Prover_, y__]]:=SupportedProperties[Prover];
```

Cascade - Lazy Thinking

```
Clear[CascadeLT];

CascadeLT[Prover_, ConjectureGenerator_, nS_, auxNS_, kConjectures_, nConjectures_]
[g_•lf, kb_•asml, userBui_, bui_, properties_, opts___?OptionQ,
{Transformer_, {tOpts___}}, {Displayer_, {sOpts___}}]:=
Block[
```

```
    {proof-object,
    conjecture,
    conjFormula,
    pv,
    newKb,
    tempNConjectures,
    tempKConjectures,
    flatConjecture,
    newNConjectures,
    currentNewConjectures = •asml[]},



(*———-Initialization———— *)
(*make sure that the list of conjectures are 'flat' so that they can be processed*)
tempKConjectures = kConjectures;
If[Head[kConjectures]=!=•asml,
    kConjectures = FlattenKB[ToKB[tempKConjectures, Info["Knowledge Base"][[1]] ]]];

tempNConjectures = nConjectures;
If[Head[nConjectures]=!=•asml,
    nConjectures = FlattenKB[ToKB[tempNConjectures, Info["Knowledge Base"][[1]] ]]];



(*Start Current Exploration Round*)
proof-object = Prover[g, kb, userBui, bui, properties, opts];


If[GetRootProofValue[proof-object]==="proved",
(*If the proof is successful, display it...*)
    Displayer[ProofSimplifier[proof-object, branches → Proved, tOpts], sOpts];
    Print["\n LAZY THINKING ::::: The proof is completed!!!!"];
    Return[{"proved", kb, nS, auxNS, kConjectures, nConjectures}],

(*...else, display the failed attempt and generate conjectures.*)
Displayer[proof-object, sOpts];
conjecture = ConjectureGenerator[proof-object, nS, auxNS, nConjectures]];

(*——- Analysis of generated conjectures ——-*)

(*If no conjecture can be generated, failure*)
If[conjecture==="nothing",
    Print["\n No conjecture was generated, Lazy Thinking failed!"];
```

```
    Return[{"failed", kb, nS, auxNS, kConjectures, nConjectures}]];



newKb = kb;
newNConjectures = nConjectures;

(*Check the conjecture formulae, whether they are new*)
For[i = 1, i ≤ Length[conjecture[[4]]], i++,
    flatConjecture = FlattenKB[conjecture][[i]];

(*If the conjecture is the same as the previous, modulo renaming of variables, failure.*)
If[(flatConjecture[[2]]/.•var[__] → "dummy")===(Last[kb][[2]]/.•var[__] → "dummy"),
    Print["No new conjecture can be derived, we are stuck, FAILURE."];
    Return[{"failed", newKb, nS, auxNS, kConjectures, newNConjectures}]
];
(* End of conjecture analysis.*)


(*update the lists of conjectures*)
AppendTo[newKb, flatConjecture];
AppendTo[newNConjectures, flatConjecture];
AppendTo[currentNewConjectures, flatConjecture];
];

Print["LAZY THINKING::::: The proof fails.
\n After analysing the failing proof, the following conjecture(s) is(are)
    added to the knowledge base: \n ",
currentNewConjectures/.•asml → Sequence,
"\n Now attempt the proof with the updated knowledge base. "];


(*the recursive call*)
CascadeLT[Prover, ConjectureGenerator, nS, auxNS, kConjectures, newNConjectures]
[g, newKb, userBui, bui, properties, opts, {Transformer, {tOpts}}, {Displayer, {sOpts}}]
];

End

End[];

HideSymbols[ProtectedSymbols[$Context]];

EndNewPackage[$Context];
```

3.3.4 **UserLanguage** Package Modification

The role of the *Theorema* **UserLanguage** package is to handle the user language commands (such as **Prove**, **Compute**, **Solve**).

Now considering **Prove**,

$$\mathbf{Prove}[P, \mathbf{using} \rightarrow Kb, \mathbf{by} \rightarrow \mathit{Prover}, \mathbf{ProverOptions} \rightarrow \{\ldots\}],$$

where *Prover* is usually one of the user provers available in the system.The user expects a proof, therefore in the **UserLanguage** package, the implementation of the **Prove** command for this case calls $\mathit{Prover}[P, Kb, \ldots]$, which then produces a proof object (using the *Theorema* proof search procedure).

However, in the case of the lazy thinking cascade, where a whole exploration cycle is carried out, the user prover, which is an argument of **CascadeLT**, is called directly in the exploration cascade, and the **UserLanguage** interface should be skipped. Therefore, for the case where the *Prover* is **CascadeLT**, the behaviour of the **Prove** command has to be changed. This is done in a function called from **Prove**, **ProveSequentially**, as follows:

```
ProveSequentially[
{f_•lf, asm_•asml, userBui_, bui_, properties_, UserProver_, {pOpts___}},
{Transformer_, {tOpts___}}, {Displayer_, {sOpts___}}, _ : False]:=
Module[{proofObj},
  If[
    StringMatchQ[ToString[UserProver], "Cascade[*]"]||
    StringMatchQ[ToString[UserProver], "CascadeLT[*]"],

    (* if the prover is Cascade or CascadeLT call them directly *)
    UserProver[f, asm, userBui, bui, properties, pOpts, {Transformer, {tOpts}},
    {Displayer, {sOpts}}],

    (* else we have a normal UserProver *)
    If[Transformer===Identity,
        proofObj = UserProver[f, asm, userBui, bui, properties, pOpts],
        proofObj = Transformer[UserProver[f, asm, userBui, bui, properties, pOpts],
          tOpts]];
    $TmaReturnedObject = proofObj;
    Displayer[proofObj, sOpts];
    proofObj]]
```

3.4 Failure Analysis: **FailureAnalyzer**

3.4.1 Description for the User

For the "normal" users, i.e. those who will call the lazy thinking implementation **CascadeLT** as a black box method to carry out some exploration rounds, the implementation of failure analysis is not supposed to be available directly, rather it is called as part of the conjecture generation phase (which, in turn, is not intended for this type of users either).

However, it has become apparent to this author — during the work of implementation, as well as while investigating the case studies — that it is useful to have access to the failure analysis phase.

This is intended for advanced users and *Theorema* developers, as debug tools. The call of these functions is not part of the *Theorema* user language, and understanding the output requires understanding of the internal representation of knowledge.

The following are available as debug/advanced failure analysis tools (typical calls):

- **FailureAnalyzer[$TmaProofObject]** will output

  $$\{goal, temporaryknowledgelist\}.$$

  This is what will be the input for the conjecture generator.
- **PrettyFailurePrinter[$TmaProofObject]** is a pretty printer for **FailureAnalyzer**. It adds to the level of verbosity, making it easier to parse the output.
- **DebugShowLeafGoals[$TmaProofObject]** will give the result of the analysis of proof situations for all the leaves of the proof tree. This function can be used to inspect failure situations that will be discarded by our analysis strategy (first-deepest).

Note that **$TmaProofObject** is a system variable that holds the current proof object. The above commands will be issued following a proof attempt, therefore we presented their most likely usage.

3.4.2 Implementation: Informal

Failure analysis is part of the conjecture generation step in the lazy thinking method, and is implemented as the package

Theorema`Provers`ConjectureGenerator`FailureAnalyzer`.

The package contains 4 functions:

- **FailureAnalyzer**, the main function that selects a failing proof situation from the proof object, according to the first deepest strategy, and returns the result,

- **AnalyzeProofSituation**, where the analysis of a proof situation is carried out, and the debug functions
- **PrettyFailurePrinter**,
- **DebugShowLeafGoals**.

The functions where analysis is being carried out call functions that work on the proof object, which are available in the package **Theorema`Provers`Common`ProofObject`**.

FailureAnalyzer:

INPUT: a proof object representing the failed proof attempt;
OUTPUT: a list of pairs **{goal, temporary knowledge}** which will be used to generate the conjectures;
USES:
- **ListEndSteps**, from the **ProofObject** package;
- **GetProofSituationAtPos**, from the **ProofObject** package;
- **AnalyzeProofSituation**.

USED BY: GenerateConjectures, in the **ComjectureGenerator** package.
SIDE EFFECTS: -.
SYNOPSIS:
- extract from the proof object the positions of the failed leaf nodes,
- apply the depth-first selection strategy, i.e. select only those proof situations that are "deep" enough,
- for each of the selected situations, analyze it (calling **AnalyzeProofSituation**),
- eliminate repetitions, return the list of candidate pairs.

AnalyzeProofSituation:

INPUT:
- a pair **{proof situation, proof value}**,
- a proof object (from which the above pair is taken);

OUTPUT: a conjecture skeleton: **{goal, temporary knowledge}**;
USES: the option variable **$TmaFAUseUnivQuantif**;
USED BY: FailureAnalyzer
SIDE EFFECTS: -
SYNOPSIS:
- Temporary knowledge (unfiltered, raw) is given by the complement between the initial knowledge and the failing proof situation;
- **If $TmaFAUseUnivQuantif** is `True`, filter out the universal implications (these can be used for knowledge rewriting) from the temporary knowledge, as well as existential formulae;
- **Else** filter out all non-ground formulae;
- Return **{failing goal, temporary (filtered) knowledge}**.

DebugShowLeafGoals:

INPUT: a proof object representing the failed proof attempt;
OUTPUT: a list of pairs **{goal, temporary knowledge}** corresponding to the failed proof situations on the leaves of the proof tree;
USES:

- **ListEndSteps**, from the **ProofObject** package;
- **GetProofSituationAtPos**, from the **ProofObject** package;
- **AnalyzeProofSituation**.

USED BY: advanced users/system developers, for debugging;
SIDE EFFECTS: -.
SYNOPSIS:

- Extract from the proof object the positions of the leaf nodes (proof situations),
- For each of the selected situations, analyze it (calling **AnalyzeProofSituation**),
- Collect all the results and return them.

PrettyFailurePrinter:

INPUT: the proof object corresponding to the failed proof;
OUTPUT: the output of the **FailureAnalyzer** in a nice way;
USES: **FailureAnalyzer**;
USED BY: advance users for debugging purposes;
SIDE EFFECTS: -;
SYNOPSIS: prints the output of the **FailureAnalyzer** in a nice way.

3.4.3 Implementation: Code

FailureAnalyzer

Package Description

```
Needs["Theorema`"]

NewPackage["Theorema`Provers`ConjectureGenerator`FailureAnalyer`",
    {"Theorema`Provers`Common`ProofObject`"}];

ProtectedSymbols[$Context] =
    Hold[FailureAnalyzer, PrettyFailurePrinter,
     DebugShowLeafGoals, FAUseUnivQuantif, $TmaFAUseUnivQuantif];

ExportedSymbols[$Context] = Join[Hold[], ProtectedSymbols[$Context]];
```

Implementation

```
Begin["`Private`"];
```

Options of the Failure Analyzer

```
Options[FailureAnalyzer] = {FAUseUnivQuantif → True};
ProcessOptions[FailureAnalyzer, opts___?OptionQ]:=
    ({$TmaFAUseUnivQuantif} = {FAUseUnivQuantif}/.{opts}/.Options[FailureAnalyzer]);
```

AnalyzeProofSituation

```
Clear[AnalyzeProofSituation];

AnalyzeProofSituation[arg1 : {psit_, pvalue_}, pobj_]:=
Block[{tempKnowledge, relTempKnowledge},
(* Select the temporary knowledge, unfiltered *)
    tempKnowledge = Complement[psit[[2]], pobj[[1, 3]]];

(* Filter the temporary knowlegde *)
    If[$TmaFAUseUnivQuantif,
    relTempKnowledge =
Select[tempKnowledge, MatchQ[#, •If[_, TM ForAll[_, _, TM Implies[_, _]]||TM Exist[___], _]]&],
    relTempKnowledge = Select[tempKnowledge, FreeQ[#, •var, ∞]&]];

(* Return the conjecture skeleton *)
    {psit[[1]], relTempKnowledge}
];
```

FailureAnalyzer

```
Clear[FailureAnalyzer];

FailureAnalyzer::usage =
"[proofObject] \n
\t Call this function to get the relevant failing proof situations from a proof";

FailureAnalyzer[pobj_]/;IsProofObject[pobj]:=
Block[{endPos,
    failingPos,
```

```
      PSitLeaves,
      PSitLeavesFailed,
      sameDepthPositions},

      (* Select from the proof object the failing positions *)
      endPos = ListEndSteps[pobj];
      failingPos = Select[endPos, (GetProofValueAtPos[pobj, #, "InternalPosition"]==="failed")&];

      (* Apply the selection strategy *)
      sameDepthPositions = Select[failingPos, (Length[First[failingPos]] == Length[#])&];

      PSitLeaves = {GetProofSituationAtPos[pobj, #, "InternalPosition"],
       GetProofValueAtPos[pobj, #, "InternalPosition"]}&/@sameDepthPositions;

      PSitLeavesFailed = Cases[PSitLeaves, {_, "failed"}];

      (*Analyze each of the selection situations, return distinct results *)
      Union[AnalyzeProofSituation[#, pobj]&/@PSitLeavesFailed]
];
```

DebugShowLeafGoals

```
Clear[DebugShowLeafGoals];


DebugShowLeafGoals::usage = "[proofObject] \n
\t Call this function to get the relevant leaf nodes proof situations from a proof";


DebugShowLeafGoals[pobj_]/;IsProofObject[pobj]:=
Block[
{failingPos,
PSitLeaves,
PSitLeavesFailed,
leafRelevantPsit,
proofTreeDepthList,
res},

(* Get all the leaf nodes *)
      failingPos = ListEndSteps[pobj];
      PSitLeaves = {GetProofSituationAtPos[pobj, #, "InternalPosition"],
GetProofValueAtPos[pobj, #, "InternalPosition"]}&/@failingPos;

(* Analyze each of these nodes *)
```

```
    leafRelevantPsit = AnalyzeProofSituation[#, pobj]&/@PSitLeaves;

(* Collect the results *)
    proofTreeDepthList = (Length[#]/2)&/@failingPos;
    res = {};
    For[i = 1, i < Length[proofTreeDepthList],
    i++,
    res = Append[res, {leafRelevantPsit[[i]], proofTreeDepthList[[i]]}]];

    res
];


PrettyFailurePrinter

Clear[PrettyFailurePrinter];


PrettyFailurePrinter[probj_]:=
Block[{},
    Print["The Proof of ::", probj[[1, 2, 1]], ":: gets stuck at \n"];
    Print["\t proving: ", #[[1, 1]], ": ", #[[1, 2]],
" with the temporary assumptions ", #[[2]], "\n"]&/@FailureAnalyzer[probj]
];


End

End[];
HideSymbols[ProtectedSymbols[$Context]];
EndNewPackage[$Context];
```

3.5 Conjecture Generation: GenerateConjectures

3.5.1 Description for the User

For the user of the *Theorema* system, the conjecture generator (**GenerateConjectures**) is called with the **Prove** command that sets up the lazy thinking exploration process:

Prove[P, *using* $\rightarrow$ Kb,
    by $\rightarrow$ $CascadeLT[Pr, GenerateConjectures, nS, auxNS, kConj, nConj]$,
    ProverOptions $\rightarrow$ {...}].

However, for advanced users, the implementation also allows direct calls to

ConjectureGenerator. The user can call the help command in *Theorema* which informs what parameters our implementation expects:

?GenerateConjectures

[proof object, desired symbol, list of auxiliary symbols, list of conjectures]
generates conjectures in a Lazy Thinking exploration.

- *proof object typically corresponds to a failed proof attempt,*
- *desired symbol denoted the function symbol that is being synthesized,*
- *the list of auxiliary symbols is taken from the algorithm scheme (Lazy Thinking Cascade call).*

The output will be the internal representation of the conjecture generated by the implementation.

3.5.2 Implementation: Informal

The **ConjectureGenerator** package contains several functions, described in detail in the following, implementing the functionality of generating conjectures from the output of the failure analysis phase (using the **FailureAnalyzer** package). Other packages needed are those implementing syntax manipulation (tuples, operators), manipulation of formula manipulation, matching algorithms (needed for the case where we allow universal formulae in the temporary knowledge).

The main functions of the package — those which implement certain functionality in the conjecture generation process — are presented in detail. Other auxiliary functions which are very simple will be mentioned and/or described briefly in the appropriate synopsis (describing the function from which these are called). They can be inspected in the complete code included in the next Subsection 3.5.3.

ConjectureGenerator

INPUT:

- a proof object (corresponding to the failed proof attempt),
- newSymbol (the new symbol to be synthesized),
- auxNSymbols (the auxiliary unknown symbols used in the algorithm scheme to define the new symbol),
- nConjectures, the current list of conjectures about the unknown symbols;

OUTPUT:

- the conjecture corresponding to the failed proof attempt;

USES:

- **FailureAnalyzer**,
- **SqnsConjectureGeneration**;

USED BY:

- **CascadeLT**;

SIDE EFFECTS: -
SYNOPSIS:

- Call **FailureAnalyzer** to get the conjecture skeletons,
- On each skeleton call **SqnsConjectureGeneration** to get the corresponding conjecture,
- Choose how many of the obtained conjectured are used - the current implementation uses the *first failed strategy*, see Chapter 2 (2.2.5), but this can be modified if a different approach is considered.

SqnsConjectureGeneration

INPUT:

- the failure situation (conjecture skeleton),
- newSymbol,
- auxNSymbols,
- list of current conjectures about the new variables;

OUTPUT:

- (typically) a formula (conjecture) corresponding to the conjecture skeleton, or
- "nothing" (string) if no conjecture can be generated (see SYNOPSIS),

USES:

- **HandleFailureUnivQuantification**,
- **AbfConstantAnalyzer**,
- **GetABFConstants**,
- **MakeRange**
- **ExtractConjectureRedundant**
- **ConjectureWithQuantifiedFailures**.

USED BY:

- **ConjectureGenerator**

SIDE EFFECTS:
SYNOPSIS:

This function implements the main functionality of the conjecture generation stage: it takes a conjecture skeleton, and constructs a formula in the internal *Theorema* representation:

$$\bullet lma[\text{"specification"}, \bullet range[], True, \bullet flist[\bullet lf[\text{"1"}, {}^{TM}ForAll[\bullet range[\ldots r \ldots], cond, {}^{TM}Implies[knowledge, goal]]]]]$$

i.e. the *Theorema* formula (conjecture, specification for the unknown functions):

$$\text{Lemma}[\text{"specification"}, \underset{cond}{\overset{\forall}{r}} (knowledge \Rightarrow goal)\ \text{"1"}],$$

where

- r is the range of variables that occur in the formula (they are collected in the conjecture generation phase from the generalization steps)
- *cond* contains additional conditions on variables (this will be, however, $True$ in general),
- *knowledge* represents the formula corresponding to the generalization of the temporary knowledge collected during the failed proof attempt,
- *goal* represents the generalization of the failing goal.

The conjecture generation implementation constructs each of the ingredients and assembles them together into the conjecture formula (which will be used as a lemma in the next step of exploration).

The conjecture generation proceeds as follows:

- First, call **HandleFailureUnivQuantification**, changing all the universal quantifiers in the skeleton into existential quantifiers. This only has effect in the case when we allow such universal quantifiers. The purpose of this is to "protect" these quantified formulae from the effect of applying the generalization strategies later on.
- Call **AbfConstantAnalyzer** with the goal as an argument. This yields the generalization substitutions (for terms and arbitrary but fixed constants).
- **If** the set of substitutions is empty, then
  - Check whether the failing goal is an explicit definition of the unknown symbol (e.g. an explicit definition of a constant),
  - Otherwise conjecture generation fails, returning **"nothing"** (the string).
- **Else** (i.e. generalization substitutions are produced) start building up the formula corresponding to the conjecture: the formula, the variable range and the conditions:
- For the inside formula (implication) $knowledge \Rightarrow goal$:
  - Select the substitutions available (for the goal), then apply them, first term substitutions, then variable substitution,
  - Same goes for the knowledge part of the implication, but here perform further filtering: eliminate those formulae that have no connections with the goal (i.e. contain arbitrary but fixed constants not occurring in the goal - these are obtained by calling **GetABFConstants**), or eliminate from the temporary knowledge implications that match known conjectures (from the list of known conjectures, **nConjectures**)- this is done by calling **ExtractConjectureRedundant**,
  - At this stage a preliminary form of the implication is available, and it may be further simplified when constructing the range/condition, or if the universal quantification generalization strategy that is employed.
- For the range, just collect the variables obtained by generalization (the appropriate range is built through a call to **MakeRange**),
- For the condition, collect the condition information (available in the output of **AbfConstantAnalyzer**), then simplify the temporary knowledge (eliminate duplicated appearing both in the condition and on the lhs of the implication),
- At this point, all the ingredients for a conjecture are available: this is a con-

jecture obtained by the application of the first two generalization strategies: term generalization and variable generalization.

- Call **ConjectureWithQuantifiedFailures** to handle the modifications that may occur if the user also wants to try the generalization strategy that handles the presence of universally quantified formulae in the temporary knowledge (the goal and temporary knowledge will be updated accordingly, see the function description, also the description of the strategy, Subsection 2.2.5).
- Following this call, assemble the conjecture: (after reversing the technical trick from the first step, if needed), and return the conjecture.

AbfConstantAnalyzer

INPUT:

- a formula (goal) corresponding to the conjecture skeleton,
- the new symbol corresponding to the function to be synthesized,
- the list of auxiliary symbols (from the algorithm scheme);

OUTPUT:

- a set {term substitutions, variable substitutions}, i.e. the generalization substitutions to be used in the assembly of the conjecture,

USES:

- **GetNewSymbolTerms**,
- **GetABFConstants**,
- **MakeTermSubstitutionRule**,
- **MakeABFSubstitutionRule**;

USED BY:

- **SqnsConjectureGeneration**;

SIDE EFFECTS:

SYNOPSIS:

This is just an interface to the process of:

- collecting the terms:

  GetNewSymbolTerms: locates all the terms of the form

  $$newSymbol[\ldots, auxSymbol[\ldots], \ldots]$$

  in the skeleton (goal);

  and arbitrary but fixed symbols from the skeleton

  GetABFConstants: locates all the arbitrary but fixed variables ($\bullet\boldsymbol{fix}$) in the skeleton (goal). It distinguishes between "normal" variables and particular representations for tuples (built in *Theorema*). If other built-in knowledge is to be used, this should be made explicit at this point;

  and
- generating the corresponding generalization rules from these sets of terms and formulae by calling:

MakeTermSubstitutionRule: takes a term as argument and returns a set

$$\{substitution, rangeForm, conditionForm\},$$

where
- $substitution$ is of the form $term \rightarrow variable$,
- $rangeForm$ describes how the respective new variable will appear into the range,
- $conditionForm$ describes additional condition for the range. Built-in domain information is included in our representation, for now concerning tuples (represented in terms of sequence variables). For other built-in domains, this function will have to be modified.

MakeABFSubstitutionRule: takes an arbitrary but fixed variable and returns a set:

$$\{substitution, rangeForm, conditionForm\},$$

where
- $substitution$ is of the form $abf \rightarrow variable$,
- $rangeForm$ describes how the respective new variable will appear into the range,
- $conditionForm$ describes additional condition for the range. Like above, various constant "types" are considered (built-in tuples expressed with sequence variables) and if more built-in knowledge is used, then more instances of this function must be constructed.

ConjectureWithQuantifiedFailures

INPUT:
- **rhs**, corresponding to the goal,
- **lhs**, corresponding to the temporary knowledge,

The input arguments are formulae that contain variables (they represent the conjecture skeleton after the application of term and variable generalizations).

OUTPUT:
- $\{newRhs, newLhs, lhsOtherGoalMatchingList, quantFlag\}$, where
  - $newRhs, newLhs$ consist of the updated skeleton,
  - $lhsOtherGoalMatchingList$ contains the list of formulae from the lhs that match the goal (rhs) - they will be deleted from the final form of the conjecture,
  - $quantFlag$ is either "Quantification" or "NoQuantification"

USES:
- **Matching** from the **Theorema‘Provers‘PLM‘Unification‘** package;

USED BY:
- **SqnsConjectureGeneration**;

SIDE EFFECTS:
SYNOPSIS:

- Look for formulae in the lhs that are existentially quantified:
- **If** there are none, done, no quantification was initially in the temporary knowledge,
- **Else** the selected formulae are those that are candidates for the mathching with the goal,
- Check which of the selected formulae match the goal, by calling **Matching** from the **PLM'Unification** package,
- **If** no such formulae exist, return the input arguments, with the flag "NoQuantification",
- **Else** select the matching formula,
- From the resulting substitutions, distinguish:
  - simple substitutions of the form $var_1 \rightarrow var_2$,
  - complex substitutions of the form $var_1 \rightarrow term$
- Replace the goal with an existential formula where the range is constructed from the left hand side of the substitutions, which are transformed in equalities (take their conjunction if several are produced),
- After replacing the old goal with the new (existential) goal, and eliminating from the temporary knowledge the formula that was matched, return them (together with the flag "Quantification"), with the rest of the goals that match (which will be eliminated from the final formula in **SqnsConjectureGeneration**).

3.5.3 Implementation: Code

Conjecture Generator

Package Description

```
Needs["Theorema`"]

NewPackage["Theorema`Provers`ConjectureGenerator`ConjectureGenerator`",
{"Theorema`Provers`ConjectureGenerator`FailureAnalyser`",
"Theorema`Language`Syntax`Core`",
"Theorema`Language`Syntax`Operator`",
"Theorema`Language`Syntax`Tuple`",
"Theorema`Language`General`HandleFormulae`",
"Theorema`Provers`Common`HandleFormulae`",
"Theorema`Provers`Common`ProofObject`",
"Theorema`Provers`PLM`Unification`"}];

ProtectedSymbols[$Context] =
Hold[
      GetABFConstants,
```

```
        GetNewSymbolTerms,
        AbfConstantAnalyzer,
        SqnsConjectureGenerator,
        GenerateConjectures,
        $ExplorationSyntacticSugar,
        ConjectureWithQuantifiedFailures,
        HandleFailureUnivQuantification];
ExportedSymbols[$Context] = Join[Hold[], ProtectedSymbols[$Context]];
```

Begin

```
Begin["`Private`"];
$ExplorationSyntacticSugar = {};
```

Auxiliary functions for SqnsConjectureGenerator

GetABFConstants

```
Clear[OccurenceCondition];
OccurenceCondition[set_, x : •fix[__]]:=
        Block[{candidates},
        candidates = Cases[set, •fix[__]];
        If[set=!={},
        Or[(x===#)&/@candidates],
        False
        ]
];
Clear[GetABFConstants];
GetABFConstants[arg_]:=
Block[
        {unemptyTuples,
        tupleConsts,
        tuples,
        ordConsts,
        res},
unemptyTuples =
Union[
Cases[arg, TM Tuple[•fix[__], •fix[__], •seq[•fix[__]]], ∞],
Cases[arg, TM Tuple[•fix[__], •seq[•fix[__]]], ∞]
];
```

```
tupleConsts = Union[Cases[arg, x : TMTuple[•seq[•fix[__]]], ∞]];

tuples = Union[unemptyTuples, tupleConsts];

ordConsts = Union[Cases[arg, x : •fix[__]/;¬OccurenceCondition[tuples, x], ∞]];

res = Join[tupleConsts, unemptyTuples, ordConsts]
];
```

GetNewSymbolTerms

```
Clear[testAuxSymbol];

testAuxSymbol[termList_, x_]:=Select[termList, ¬FreeQ[#, x, ∞]&];

Clear[GetNewSymbolTerms];

GetNewSymbolTerms[goal_, newSymbol_, auxNewSymbols_]:=
Block[{termList,
      res},

termList = Cases[goal, newSymbol[__], ∞];

res = testAuxSymbol[termList, #]&/@auxNewSymbols;

Union@@res
];
```

SubstitutionRules

```
Clear[MakeTermSubstitutionRule];

MakeTermSubstitutionRule[term_]:=
Block[{varId, res},
      varId = NewUpperCaseIdentifier[];
      res = {term → •var[varId], •var[varId], ToExpression["TMIsTuple"][•var[varId]]}
];

Clear[MakeABFSubstitutionRule];

MakeABFSubstitutionRule[arg : TMTuple[x1_, x2_, x3 : •seq[__]]]:=
      Block[{varId,
      res},
      varId = NewUpperCaseIdentifier[];
```

```
        res = {arg → •var[varId],
        ToExpression["TMIsTuple"][•var[varId]],
        TMAnd[TMNot[ToExpression["is−−empty−−tuple"][•var[varId]]],
TMNot[ToExpression["is−−singleton−−tuple"][•var[varId]]]]}
];
MakeABFSubstitutionRule[arg : TMTuple[x1_, x2 : •seq[__]]]:=
        Block[{varId,
                res},

varId = NewUpperCaseIdentifier[];

res = {arg → •var[varId],
ToExpression["TMIsTuple"][•var[varId]],
TMNot[ToExpression["is−−empty−−tuple"][•var[varId]]]}
];
MakeABFSubstitutionRule[arg : TMTuple[x1 : •seq[__]]]:=
        Block[{varId,
                res},
varId = NewUpperCaseIdentifier[];

res = {arg → •var[varId],
ToExpression["TMIsTuple"][•var[varId]],
"nothing"}
];
MakeABFSubstitutionRule[arg : TMTuple[x1 : •seq[__]]]:=
        Block[{varId,
                res},
varId = NewUpperCaseIdentifier[];
res = {arg → •var[varId], •var[varId],
ToExpression["TMIsTuple"][•var[varId]]}
];
MakeABFSubstitutionRule[arg : TMTuple[x1 : •seq[__]]]:=
        Block[{varId,
                res},
varId = NewLowerCaseIdentifier[];
res = {arg → TMTuple[•var[•seq[varId]]],
•var[•seq[varId]],
"nothing"}
];
Clear[NewNiceIdentifier]
NewNiceIdentifier[arg_Symbol]:=
```

```
    Block[{uniqueIntermediaryId,
        idLength,
        argLength = StringLength[ToString[arg]],
        idString,
        zeroAccumulator,
         zeroes = "",
        finalIdString},

(* idLength represents the length
allocated for the number following the symbol to the variable
set it higher if more than 99 variables are expected
to pop up as result of generalization. *)

idLength = 2;


uniqueIntermediaryId = ToString[Unique[ToString[arg]]];

zeroAccumulator = idLength - (StringLength[uniqueIntermediaryId] - argLength);

While[(zeroAccumulator = zeroAccumulator - 1) ≥ 0, zeroes = StringInsert[zeroes, "0", 1]];

If[StringLength[uniqueIntermediaryId] - argLength < idLength,
    finalIdString = StringInsert[uniqueIntermediaryId, zeroes, argLength + 1],
    finalIdString = uniqueIntermediaryId
];
ToExpression[finalIdString]
]


MakeABFSubstitutionRule[arg : •fix[_]]:=
    Block[{res},
    varId = NewNiceIdentifier[arg[[1]]];
    res = {arg → •var[varId], •var[varId], "nothing"}
];


AbfConstantAnalyzer

Clear[AbfConstantAnalyzer];

AbfConstantAnalyzer[goal_, newSymbol_, auxNSymbols_]:=
Block[{nsTerms,
    abfVr,
    substNsTerms,
```

```
        substAbfVr,
        res},

nsTerms = GetNewSymbolTerms[goal, newSymbol, auxNSymbols];
abfVr = GetABFConstants[goal];
substNsTerms = MakeTermSubstitutionRule[#]&/@nsTerms;
substAbfVr = MakeABFSubstitutionRule[#]&/@abfVr;
res = {substNsTerms, substAbfVr}
];
```

Handling of variables in failed proof situations

HandleFailureUnivQuantification

```
Clear[HandleFailureUnivQuantification];

HandleFailureUnivQuantification[failureSit_]:=
        Block[{newFailureSituation},
        newFailureSituation = failureSit/.{TM ForAll → TM Exists}
        ];
```

Constructing the conjectures when the failure situation contains quantified formulae

ConjectureWithQuantifiedFailures

```
Clear[ConjectureWithQuantifiedFailures];

ConjectureWithQuantifiedFailures[rhs_, lhs_]:=
Block[{quantifiedFailures,
                goalMatchingList,
                simpleSubstitutions,
                complexSubstitutions,
                matchingSubstitution,
                newInnerFormula,
                resLhs,
                resRhs,
                temp,
                existentialRange,
                varEx,
                varGen,
                lhsOtherGoalMatchingList},


quantifiedFailures = Select[lhs, ¬FreeQ[#, TM Exists]&];
```

```
If[Length[quantifiedFailures] == 0,
      Return[{rhs, lhs, •asml[], "NoQuantification"}]];

goalMatchingList = DeleteCases[Map[Matching[rhs, #[[3]]]&, quantifiedFailures], False];

If[Length[goalMatchingList] == 0,
Return[{rhs, lhs, •asml[], "NoQuantification"}]];

lhsOtherGoalMatchingList =
Select[Complement[lhs, quantifiedFailures], (Matching[rhs, #]=!=False)&];

matchingSubstitution = goalMatchingList[[1]];

simpleSubstitutions = Select[matchingSubstitution, MatchQ[#, •var[_] → •var[_]]&];

complexSubstitutions = Complement[matchingSubstitution, simpleSubstitutions];

Clear[temp];
temp = •asml[];

newInnerFormula =
Switch[Length[complexSubstitutions],
      1, TM Equal[complexSubstitutions[[1, 1]], complexSubstitutions[[1, 2]]],
      _, (For[i = 1, i ≤ Length[complexSubstitutions], i++,
AppendTo[temp, TM Equal[complexSubstitutions[[i, 1]], complexSubstitutions[[i, 2]]]]];
TM And[temp/.•asml → Sequence])
];

Clear[temp];
temp = •asml[];
resLhs = Switch[Length[complexSubstitutions],
      1, Select[Complement[lhs, quantifiedFailures],
¬FreeQ[#, complexSubstitutions[[1, 1]]]&],
      _, For[i = 1, i ≤ Length[complexSubstitutions], i++, AppendTo[temp, Select[
Complement[lhs, quantifiedFailures], ¬FreeQ[#, complexSubstitutions[[i, 1]]]&]]]
];

(* check if those existential variables actually occur in the formula,
if not, simplify it to exclude them *)

varGen = Union[Cases[newInnerFormula, •var[_], Infinity]];

existentialRange =
```

```
DeleteCases[quantifiedFailures[[1, 1]],
•simpleRange[•var[a_]]/;FreeQ[varGen, •var[a]]];

If[existentialRange=!=•range[],
resRhs = TMExists[existentialRange, quantifiedFailures[[1, 2]], newInnerFormula],
resRhs = newInnerFormula];

{resRhs, resLhs, lhsOtherGoalMatchingList, "Quantification"}
];
```

SqnsConjectureGenerator

```
Clear[SelectNoDuplicates];

SelectNoDuplicates[x_]:=
Block[{positions},
      positions = Position[x, #]&/@Union[x];
      duplicatePositions = Delete[#, 1]&/@(positions/.•asml → List);
      Delete[x, List/@Flatten[duplicatePositions]]
];

Clear[ExtractConjectureRedundant];

ExtractConjectureRedundant[nConjectures_]:=
Block[
      {nCFlat,
      conjGoals,
      conjInnerForm},

If[Head[nConjectures]=!=•asml,
nCFlat = FlattenKB[ToKB[nConjectures, Info["Knowledge Base"][[1]]]],
nCFlat = nConjectures];
conjInnerForm = #[[2, 3]]&/@nCFlat;
conjGoals = If[Head[#]===TMImplies, #[[2]], #]&/@conjInnerForm;
patternGoals = conjGoals/.•var[__] → __
];

Clear[MakeRange];
MakeRange[P_[•var[x_]]]:=•predRange[•var[x], P];
MakeRange[•var[x_]]:=•simpleRange[•var[x]];

Clear[SqnsConjectureGenerator];

SqnsConjectureGenerator[failureSit_, newSymbol_, auxNSymbols_, nConjectures_]:=
Block[
      {range = •range[],
```

```
        initCond = •asml[],
        cond,
        finalCond,
        initLhs,
        initRhs,
         lhs,
        finalLhs,
        finalRhs,
        finalFormula,
        rhs,
        res,
        source,
        terms,
        variables,
        filteredVariables,
        abfConstantsRhs,
        termSubstitutions,
        abfSubstitutions,
        cjctFormula,
        abfFilter,
        lLabel = StringReplace[
ToString[Unique["specification"]], "specification" → "specification "],
        env,
        tempLhs,
        tempSugaredCond,
        nConjGoalPatterns,
        arg,
        failureQuantifQ,
        preFinalLhs,
        preFinalRhs,
        finalFormula1,
        temp,
        tempASLhs,
        cjctFormula1,
        finalRange,
        finalRange1,
        lhsGoalMatchingToEliminate},

arg = HandleFailureUnivQuantification[failureSit];


source = AbfConstantAnalyzer[arg[[1]], newSymbol, auxNSymbols];

If[Union[source]=={{}},
```

```
    If[MatchQ[arg[[1, 2]], TM Equal[newSymbol[_], _]],
    cjctFormula =
•If[UniqueLabel["explicit def of" <> ToString[•[newSymbol]]], arg[[1, 2]], •finfo[]],
    cjctFormula = "nothing"],


(* extract the proper sources for constructing the conjecture *)
    terms = source[[1]];
    variables = source[[2]];
    termSubstitutions = #[[1]]&/@terms;


(* start building the formula part of the conjecture *)


    If[arg[[2]]===•asml[],
    initRhs = arg[[1, 2]]; initLhs = •asml[],
    initRhs = arg[[1, 2]]; initLhs = #[[2]]&/@ arg[[2]]
    ];
    rhs = initRhs/.termSubstitutions;


(* here collect all the •fix variables left on
'rsh' and filter the set 'variables' by elinimating all others,
in order to construct later the proper range and condition for the conjecture *)

    abfConstantsRhs =
Cases[rhs, TM Tuple[•seq[•fix[__]]]|TM Tuple[•fix[__], •fix[__], •seq[•fix[__]]]|
TM Tuple[•fix[__], •seq[•fix[__]]]|•fix[__], ∞];


    abfConstantsLhs = GetABFConstants[arg[[2]]];


    abfFilter = Complement[abfConstantsLhs, abfConstantsRhs];


    filteredVariables = variables;


(* get the substitutions from the variables information *)


    abfSubstitutions = #[[1]]&/@filteredVariables;
```

```
        preFinalRhs = rhs/.abfSubstitutions;

        lhs = initLhs/.termSubstitutions/.abfSubstitutions;


        If[FreeQ[lhs, •var, ∞],
        finalLhs = SelectNoDuplicates[Select[lhs, FreeQ[#, •fix, ∞]&]],
        finalLhs = Union[Select[lhs, FreeQ[#, •fix, ∞]&]]
        ];

(* since some simplification needs to be done after determining the condition,
the finalFormula of the conjecture is constructed then *)


(* construct the range of the conjecture *)
        If[terms=!={}, AppendTo[range, MakeRange[#[[2]]]]&/@terms];

        If[filteredVariables=!={},
AppendTo[range, MakeRange[#[[2]]]]&/@filteredVariables];


(* construct the condition of the conjecture *)
        If[filteredVariables=!={}, AppendTo[initCond, #[[3]]]&/@filteredVariables];

        cond = Union[Select[initCond, (#=!="nothing")&]];


(* eliminate from the conditions the formulae that are syntactic sugar *)

        If[$ExplorationSyntacticSugar=!={},
        tempSugaredCond = cond;
        cond = DeleteCases[tempSugaredCond,
patt : head_[__]/;MemberQ[$ExplorationSyntacticSugar, head]];
];

(* finished eliminating the syntactic sugar *)


        finalCond = Switch[Length[cond],
        0, True,
        1, cond[[1]],
        _, TM And[cond/.•asml → Sequence]
        ];
```

```
(* now that the condition is here, simplify the formula,
so that things in the condition do not appear also in the formula *)

      tempLhs = finalLhs;
      If[Head[finalCond]===TM And,
      tempLhs = Select[tempLhs, ¬MemberQ[finalCond, #]&],
      tempLhs = Select[tempLhs, finalCond=!=#&]
      ];


(* more things to get rid of from the lhs of the implication :
all implications: because they should have
been rewritten before and should play no part in the conjecture;
the goal patterns in the newConjectures list *)

      tempLhs = Select[tempLhs, Head[#]=!=TM Implies&];
      nConjGoalPatterns = ExtractConjectureRedundant[nConjectures];
      For[i = 1, i ≤ Length[nConjGoalPatterns],
i++, tempLhs = DeleteCases[tempLhs, nConjGoalPatterns[[i]]]];
      preFinalLhs = tempLhs;


(* Some more processing has to be done on the formulae to handle the situation
when quantified formulae appear in the failing proof situation *)

      {finalRhs, finalLhs, lhsGoalMatchingToEliminate, failureQuantifQ} =

      ConjectureWithQuantifiedFailures[preFinalRhs, preFinalLhs];

      If[failureQuantifQ==="Quantification",
      Clear[temp];
      temp = •asml[];
      For[i = 1, i ≤ Length[auxNSymbols], i++, temp = Join[temp,
Select[Complement[preFinalLhs, finalLhs], ¬FreeQ[#, auxNSymbols[[i]]]&]]];
      tempASLhs = Complement[DeleteCases[temp, TM Exists[__]],
lhsGoalMatchingToEliminate];

      finalFormula1 = Switch[Length[tempASLhs],
      0, "none",
      1, tempASLhs[[1]],
      _, TM And[tempASLhs/.•asml → Sequence]],
      finalFormula1 = "none"
      ];
```

```
    If[finalFormula1==="none" ∧ failureQuantifQ==="NoQuantification",
finalLhs = DeleteCases[finalLhs, TM Exists[___]]];

    finalFormula = Switch[Length[finalLhs],
    0, finalRhs,
    1, TM Implies[finalLhs[[1]], finalRhs],
    _, TM Implies[TM And[finalLhs/.•asml → Sequence], finalRhs]
    ];

    If[finalFormula1==="none" ∧ failureQuantifQ==="NoQuantification",
finalFormula = finalFormula/.{TM Exists → TM ForAll}];


(* assemble the conjecture *)
    finalRange = Intersection[range,
(•simpleRange/@Union[Cases[finalFormula, •var[_], ∞]]/.List → •range)];
    cjctFormula = •If[ToString[NewIdentifier["1"]],
TM ForAll[finalRange, finalCond, finalFormula]];


    If[finalFormula1=!="none",
    finalRange1 = Intersection[range,
(•simpleRange/@Union[Cases[finalFormula1, •var[_], ∞]]/.List → •range)];
cjctFormula1 = •If[ToString[NewIdentifier["2"]],
TM ForAll[finalRange1, finalCond, finalFormula1]]
]];


If[cjctFormula==="nothing",
"nothing",
env = EmptyEnvironment[•lma, lLabel];
AppendTo[env[[4]], cjctFormula];
If[finalFormula1=!="none", AppendTo[env[[4]], cjctFormula1]];
env
]];
```

GenerateConjectures

```
Clear[GenerateConjectures];

GenerateConjectures[prObj_, newSymbol_, auxNSymbols_, nConjectures_]:=
Block[
```

```
        {failures,
        res},
        failures = ca;
        res =
        SqnsConjectureGenerator[#, newSymbol, auxNSymbols, nConjectures]&/@failures;
        res[[1]]
];
```

End

```
End[];
HideSymbols[ProtectedSymbols[$Context]];
EndNewPackage[$Context];
```

3.6 Summary of the Implementation

To summarize, we present in Figure 3.1 the functions of our implementation. Arrows represent function calls, dotted boxes represent packages (name is indicated), boxes represent functions, with the arrow returning to the same box indicating a recursive call.

3.7 Using Existing Theorema Reasoners in the Lazy Thinking Cascade

We already mentioned, both in Chapter 2 (2.2.6) and in the corresponding implementation section above, that the failure analysis and subsequent conjecture generation phases of lazy thinking play nicely together with the PC(S) proof method (generalized rewriting).

In fact, all our user provers implemented for the various case studies include the PC component. In *Theorema* the functionality of the PC method is implemented in the following packages:

- **"Theorema`Provers`QuantifierRewriting`QR`"** which contains inference rules for generalized (knowledge) rewriting, and
- **"Theorema`Provers`PCSBasicND`"** which contains predicate logic inference rules that are applied after the knowledge rewriting phase.

Therefore, these basic provers should be present in any user prover that is called to reason about a problem in the context of a lazy thinking case study in *Theorema*.

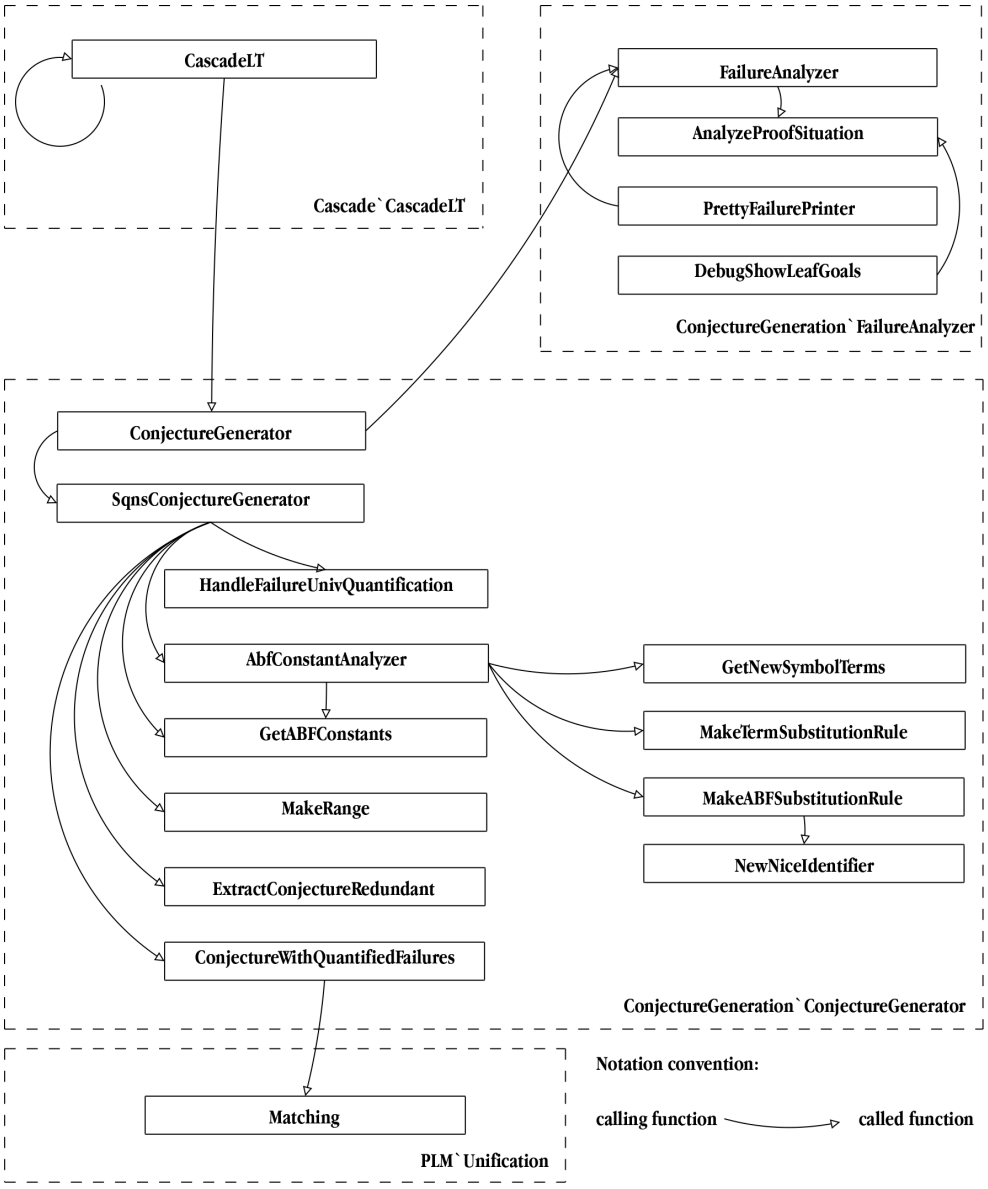

Figure 3.1: Schematic representation of the lazy thinking implementation in *Theorema*

CHAPTER 4

Synthesis of an Algorithm for Gröbner Bases

In this chapter we show how, using our lazy thinking implementation, we can solve the problem of Gröbner bases construction. For this, we formulate the problem in *Theorema*, give the knowledge base we use in the exploration process, including the Critical Pair/Completion (CPC) algorithm scheme and give the intermediary calls that lead to the generation of conjectures and proofs generated in the process.

Several technical difficulties have to be overcome: a direct induction proof of the correctness of an instantiation of the CPC algorithm scheme is not straightforward, and in fact we do not even attempt such a proof. It is well known that, for example in the case of term rewriting (Knuth Bendix), the algorithm does not even terminate.

Rather, we *preprocess* the scheme, i.e. we prove a simple property of the algorithm scheme (which holds if the algorithm terminates). For this, we introduce an inference rule for proving invariant properties of the CPC scheme, i.e. those that hold for intermediary arguments. The properties thus proved will be used in the knowledge base in a lazy thinking exploration.

The main part of the exploration is based on Bruno Buchberger paper [Buchberger, 2004b], which outlines the synthesis process.

Our preprocessing results only hold if the algorithm terminates. The question of termination can be reduced as in literature, to the Dickson lemma.

The following 4 Sections of this chapter are printed from *Theorema* notebooks containing the case studies. This illustrates one of the strong points of the *Theorema* system. The output of the automated reasoners are nicely structured, human-readable proofs in nice syntax.

4.1 The Problem of Gröbner Bases Construction

We give an informal discussion of the **synthesis situation corresponding to the problem of Gröbner bases construction**. The full details are presented in subsequent Sections, consisting of exploration steps with the *Theorema* system.

First, load the system:

```
Needs["Theorema`"]
```

Also load packages implementing lazy thinking (cascade, failure analyzer, conjecture generator):

```
Needs["Theorema`Language`Syntax`Core`"]
```

```
Needs["Theorema`Provers`ConjectureGenerator`FailureAnalyzer`"]
```

```
Needs["Theorema`Provers`ConjectureGenerator`ConjectureGenerator`"]
```

```
Needs["Theorema`Provers`Cascade`CascadeLT`"]
```

```
$RecursionLimit = Infinity;
```

```
$TmaDefaultSearchDepth = 100;
```

4.1.1 Statement of the Problem

In the sequel, F, G denote sets of polynomials from some polynomial ring. We want to find an algorithm GB that satisfies the following *correctness theorem*:

```
Theorem["Groebner bases specification", any[F], with[is-finite[F]]
    is-finite-Groebner-basis[F, GB[F]]]
```

where:

```
Definition["is finite Groebner basis", any[F, G], with[is-finite[F]],
  is-finite-Groebner-basis[F, G] ⇔ ⋀ { is-finite[G]
                                     { is-Groebner-basis[G]
                                     { ideal[F] = ideal[G] ]
```

This is a problem formulated in polynomial ring theory. We give an informal presentation of the relevant fragments of the theory used in this case study.

4.1.2 The Underlying Theory

The case study is based on the following concepts:

- f ≺ g: "polynomial f is smaller in the Noetherian order ≺ than polynomial g",
- is-Groebner-basis[F, G]: "the polynomial set G is a Gröbner basis of polynomial set F",

− $f \longrightarrow_F g$: "polynomial f is reduced in one step to polynomial g, modulo the polynomial set F",

− is-Church-Rosser[⟶]: "the relation ⟶ has the Church Rosser property",

− $f \downarrow_F g$: "polynomials f and g have a common successor under reduction modulo polynomial set F",

− is−pp[p]: "p is a power product",

− lm[f]: "the leading monomial of polynomial f",

− lp[f]: "the leading power product of polynomial f",

− rd[g, f]: "the result of reducing polynomial g modulo polynomial f, in one step",

− rd[g, F]: "the result of reducing polynomial g modulo the polynomial set F, in one step",

− trd[g, F]: "the result of totally reducing polynomial g modulo the polynomial set F".

The symbols used for these concepts are part of the **language of the underlying theory**.

Remark (Notation). In this context, for any relation R, R_F (e.g. $\longrightarrow_F$) is used as a notation for "Currying", i.e. R_F is a notation for R[F] (e.g. $\longrightarrow_F$), defined by R[F][x, y]⟺R[F, x, y].
Now, in the theory of relations we have T[P] (transitive closure), ES[P] (exists successor), where P is a binary relation. Then, again by currying, $\downarrow_F$ stands for ES[R[F]].

The definitions and properties of these concepts, i.e. the **knowledge base of the underlying theory**, will be given in the appropriate sections below. For now consider some properties of the Noetherian ordering ≺:

Proposition["reduction is compatible with the ordering",

$$\forall_{f,g}\ rd[f,\ g] \preceq f\Big]$$

Moreover, from the definition of the reduction function rd:

Definition["reduction modulo polynomials",

$$\forall_{f,g}\left(rd[f,\ g] = \begin{cases} f - \frac{lm[f]}{lm[g]}\, g & \Leftarrow \quad lp[g] \mid lp[f] \\ f & \Leftarrow \quad \text{otherwise} \end{cases}\right)\Big]$$

it is easy to derive the following:

Proposition["strict ordering condition",

$$\forall_{f,g}\ (rd[f,\ g] \prec f) \Rightarrow (lp[g] \mid lp[f])\Big]$$

This result will be used below, in the context of the algorithm scheme.

The **inference mechanism of the underlying theory** consists of a combination of the PC method, predicate logic and rewriting (simplification). The user prover implementing the inference mechanism is loaded into the system by calling:

```
Needs["Theorema`Provers`UserProvers`BasicProver`"]
```

4.1.3 Algorithm Scheme: Critical−Pair/Completion

The Critical−Pair/Completion algorithm scheme (CPC) for polynomial ideals, mentioned in Chapter 2 (Example 7), is applicable in this case, with the instantiation is-CPC-poly-scheme[GB, lc, df], introduced in *Theorema* as:

```
Algorithm["CPC scheme", any[F, g1, g2, p̄],

  GB[F] = GB[F, pairs[F]]
  GB[F, ⟨⟩] = F

  GB[F, ⟨⟨g1, g2⟩, p̄⟩] =
    where[f = lc[g1, g2], h1 = trd[rd[f, g1], F], h2 = trd[rd[f, g2], F],
      { GB[F, ⟨p̄⟩]                                                    ⇐  h1 = h2
      { GB[F ⌢ df[h1, h2], ⟨p̄⟩ ≍ ⟨⟨F_k , df[h1, h2]⟩ | k=1,...,|F|⟩]  ⇐  otherwise ]
```

where **lc** and **df** are new (auxiliary) symbols, representing subalgorithms. Additionally, we require:

```
Proposition["lc size specification",
  ∀_{g1,g2} ⋀{ rd[lc[g1, g2], g1] ≺ lc[g1, g2]
             { rd[lc[g1, g2], g2] ≺ lc[g1, g2] ]
```

From Proposition["strict ordering condition"], we obtain immediately:

```
Proposition["lc divisibility specification",
  ∀_{g1,g2} ⋀{ lp[g1] | lc[g1, g2]
             { lp[g2] | lc[g1, g2] ]
```

Now, in order to synthesize the algorithm **GB (**the solution to the Gröbner bases problem) by lazy thinking, we need to add to the inference mechanism appropriate inference rules to prove properties of the algorithm proposed. For simple recursive schemes like the divide-and-conquer such properties can be proved by a well-founded induction rule, with the well-founded ordering established between terms in the recursive call. However, the CPC scheme is more complex, and a well-founded ordering is not straightforward to establish between the terms in the recursion.

Rather, *before* we begin the synthesis process, we prove certain properties of the algorithm scheme, *independently of the problem we want to solve*. For this, we formulate an inference rule for proving such properties. This step amounts to *preprocessing the algorithm scheme*, and the results of this step will be used in the synthesis exploration.

Representations of reduction terms

In the definition introduced above, Algorithm["CPC scheme"], the **where** construct allows specifications of local variables f, h1, h2, which make it easier to see the structure of the algo−rithm,

However, due to some technical considerations, for the in the following, we do not use the local variables, but rather provide the full term. Terms of the form

```
trd[rd[p, g1], G] = trd[rd[p, g2], G]
```

correspond to a "diamond" (confluence) diagram (Figure 4.1).

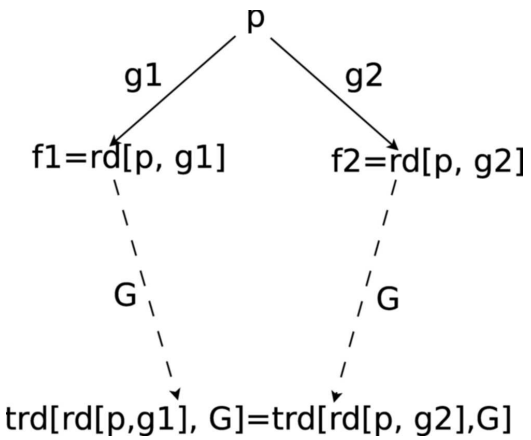

Figure 4.1

In Figure 4.1, full arrows correspond to the reduction relation, with the label showing the element used for reduction, and dashed arrows represent total reduction, with the set represented as a label to the arrow.

Such terms will show up in proofs and in the knowledge, and also in the specifications gener−ated by lazy thinking.

4.2 Preprocessing the CPC Scheme

A direct induction proof of properties of the Algorithm["CPC scheme"], our proposed solution for the Gröbner bases problem, is difficult to set up, due to the complexity of the recursive calls. Rather, we derive consequences of the schemes (we *preprocess* the scheme) and use this knowledge in the lazy thinking synthesis process.

Here we propose a generic inference rule for proving properties of the algorithm **GB**. We then specialize this inference rule to prove "invariant" properties, i.e. properties that hold for all intermediary sets of polynomials generated during the execution of the algorithm. This will significantly simplify the inference rule, as many of the cases are easily verified. If the algo−rithm terminates, then the property holds also for the final result of the algorithm.

We then show how a simple property of Algorithm["CPC scheme"] can be derived.

4.2.1 A Generic Inference Rule for Proving Properties of CPC Algorithms

Consider Algorithm["CPC Scheme"]. We propose the following inference rule:

To prove universal properties **P** that relate the result of the application of the algorithm **GB** to its input argument:

$$\forall_{F} \; P[F,\ GB[F]]$$

for some ternary predicate (property) **C**, take

$$F_0,\ G_0,\ g1,\ g2,\ \overline{p}$$

arbitrary but fixed two polynomial sets, two polynomials, and a sequence of pairs of polynomi−als, respectively, and show:

−"base case":

$$C[F_0,\ F_0,\ pairs[F_0]],$$

−"step":

Case $trd[rd[lc[g1,\ g2],\ g1],\ G_0] = trd[rd[lc[g1,\ g2],\ g2],\ G_0]$:

$$C[F_0,\ G_0,\ \langle\langle g1,\ g2\rangle,\ \overline{p}\rangle] \Rightarrow C[F_0,\ G_0,\ \langle\overline{p}\rangle],$$

Case $trd[rd[lc[g1,\ g2],\ g1],\ G_0] \neq trd[rd[lc[g1,\ g2],\ g2],\ G_0]$:

$$C[F_0,\ G_0,\ \langle\langle g1,\ g2\rangle,\ \overline{p}\rangle] \Rightarrow C\Big[F_0,\ G_0 \frown df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]],\ \langle\overline{p}\rangle \asymp \Big\langle\langle G_{0\,k}, df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]]\rangle_{k=1,\ldots,|G|}\Big\rangle\Big],$$

−"final step":

$$C[F_0,\ G_0,\ \langle\rangle] \Rightarrow P[F_0,\ G_0].$$

The arguments of the property **C** are associated with intermediary sets generated by the execu−tion of the algorithm **GB**. The first argument of **C**, F_0, plays the role of the polynomial set the algorithm **GB** is called on, the second, G_0, plays the role of intermediary sets generated by the executions of the algorithm, while the third argument is the list of pairs associated with each intermediary execution step.

Informally,

− if the property **C** is true for the initial situation (initial call of the algorithm **GB**, with the initial set F_0) − "base case" −, **and**

− assuming it is true for some intermediary set G_0, we can prove it is true for the value given by the next iteration of the algorithm (where the next iteration is given by case, depending whether the element lc[g1, g2] can be reduced in two ways by g1, g2 respectively, then totally reduced modulo the current intermediary set G_0 to the same element) − "step"−, **and**

− finally the fact that the property **C** holds for configuration where the set of pairs is empty implies that the property **P** is true for F_0 and G_0 −"final step",

then the property **P** holds for the initial set F_0 and GB[F_0], provided that GB[F_0] is an appropri−ate object (i.e. a set of polynomials). That is, the property holds **if the algorithm GB termi−nates**.

Note that if the algorithm terminates, the inference rule corresponds to an induction on the recursive calls of **GB**.

To use this inference rule, one has to find an appropriate property **C**. As mentioned above, using this inference rule will give a **proof conditioned by the termination of the algorithm GB**.

4.2.2 CPC Inference for Invariant Properties

Let **P** be some property of the algorithm **GB** as described above. Consider the following prop−erty **C**, defined for any F, G, polynomial sets, and B, sets of pairs of polynomials:

$$C[F, G, B] \Leftrightarrow (P[F, G] \wedge B \subseteq G \times G).$$

This particular choice of **C** assigns the following roles to its ingredients:

− **P** is a property that holds throughout the execution of the algorithm, i.e. every intermediary set G is in the same relation with the initial F. Stated differently, **P** is an **invariant property** of **GB**,

− B, the set of polynomial pairs associated with each intermediate call of **GB**, contains only pairs of elements of the intermediate set (pair closure).

This choice of **C** limits the range of properties which can be proved. However, at the same time, it simplifies significantly the inference rule.

To prove

$$\forall_{F}\ P[F,\ GB[F]]$$

let

$$F_0,\ G_0,\ g1,\ g2,\ \overline{p}$$

be arbitrary but fixed two polynomial sets, two polynomials, and a sequence of pairs of polyno−mials, respectively.

− "base case" becomes:

$$P[F_0,\ F_0] \wedge pairs[F_0] \subseteq F_0 \times F_0$$

The second part of the conjunction is true by a basic property of the pairs function:

```
Proposition["pairs and cartesian product", any[F],
  pairs[F] ⊆ F×F]
```

The first part of the conjunction is true because of the choice of **P**: it is an invariant property, that should hold in particular for the first set in the recursive call. Therefore, "base case" can be discarded completely.

− "step" becomes:

Case $trd[rd[lc[g1,\ g2],\ g1],\ G_0] = trd[rd[lc[g1,\ g2],\ g2],\ G_0]$:

$$P[F_0,\ G_0] \wedge (\langle\langle g1,\ g2\rangle,\ \overline{p}\rangle \subseteq G_0 \times G_0) \Rightarrow P[F_0,\ G_0] \wedge (\langle\overline{p}\rangle \subseteq G_0 \times G_0)$$

which clearly holds. This case can be completely discarded.

Case $trd[rd[lc[g1,\ g2],\ g1],\ G_0] \neq trd[rd[lc[g1,\ g2],\ g2],\ G_0]$:

$$P[F_0,\ G_0] \wedge \langle\langle g1,\ g2\rangle,\ \overline{p}\rangle \subseteq G_0 \times G_0 \Rightarrow$$
$$P[F_0,\ G_0 \frown df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]]] \bigwedge$$
$$\langle\overline{p}\rangle \asymp \left\langle \langle G_{0\,k}, df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]]\rangle \Big|_{k=1,\ldots,|G|} \right\rangle \subseteq$$
$$(G_0 \frown df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]]) \times$$
$$(G_0 \frown df[trd[rd[lc[g1,\ g2],\ g1],\ G_0],\ trd[rd[lc[g1,\ g2],\ g2],\ G_0]]).$$

In fact, although the pair closure part of this proof situation looks complex, its proof is not difficult. The algorithm **GB** says that when adding to the set G_0 the element df[…, …], its corresponding set of pairs has to be completed with pairs containing the new element df[…, …]. This clearly keeps the pairs in the cartesian product:

$$\text{Proposition}\Big["\text{completion of pairs and cartesian product}",\ any[F,\ x,\ \overline{p}],$$
$$\langle\overline{p}\rangle \subseteq F \times F \Rightarrow \langle\overline{p}\rangle \asymp \left\langle \langle F_k, x\rangle \Big|_{k=1,\ldots,|F|} \right\rangle \subseteq (F \frown x) \times (F \frown x)\Big]$$

Therefore the closure part can be discarded, and only the following has to be proved:

```
P[F0 , G0 ] ∧ ⇒ P[F0 , G0 ⌢ df[trd[rd[lc[g1, g2] , g1], G0 ], trd[rd[lc[g1, g2] , g2], G0 ] ]]
```

− "final step" becomes:

```
C[F0 , G0 ] ∧ ⟨⟩ ⊆ G0 × G0 ⇒ P[F0 , G0 ].
```

This clearly holds, so it can be discarded.

We summarize the inference rule to prove **invariant** properties of GB: To prove an invariant universal property of GB:

```
∀ P[F, GB[F]]
F
```

take

```
F0 , G0 , g1, g2,
```

arbitrary but fixed two sets of polynomials and two polynomials respectively, assume:

```
trd[rd[lc[g1, g2] , g1], G0 ] ≠ trd[rd[lc[g1, g2] , g2], G0 ]
```

and prove:

```
P[F0 , G0 ] ⇒ P[F0 , G0 ⌢ df[trd[rd[lc[g1, g2] , g1], G0 ], trd[rd[lc[g1, g2] , g2], G0 ] ]].
```

Implementation

We have implemented the invariant property proof rule above as a *Theorema* basic prover, CPCInd. This basic prover is combined with provers that provide natural style deduction (with generalized rewriting, the PC method), equality proving (rewriting) into the user prover CPCIn− duction, which is loaded into the system by calling:

```
Needs["Theorema‘Provers‘UserProvers‘CPCInduction‘"]
```

4.2.3 A Simple Property of the CPC Scheme

Using our prover, we can prove invariant properties of GB, like:

```
Proposition["GB subset", any[F],
  F ⊆ GB[F]]
```

That is, the initial polynomial set is contained in the final set (if the algorithm terminates).

To prove this proposition, we need the following property of polynomial sets:

```
Proposition["subset and append", any[F, G, x],
  (F ⊆ G) ⇒ (F ⊆ G ⌢ x)]
```

The knowledge base:

```
Theory["preprocess CPC",
  Proposition["subset and append"]
    Algorithm["CPC scheme"]]
```

The proof call:

```
Prove[Proposition["GB subset"],
    using → Theory["preprocess CPC"], by → CPCInduction] // Last
proved
```

The proof succeeds, by a simple application of Proposition["subset and append"].

Consider now the following property of the algorithm scheme:

```
Proposition["preprocessed CPC scheme", any[g1, g2, F],
 ((g1 ∈ GB[F] ∧ g2 ∈ GB[F]) ⇒
    ⋁ { trd[rd[lc[g1, g2], g1], GB[F]] = trd[rd[lc[g1, g2], g2], GB[F]]
      { df[trd[rd[lc[g1, g2], g1], GB[F]], trd[rd[lc[g1, g2], g2], GB[F]]] ∈ GB[F] ))]
```

The above is not an invariant property of the algorithm scheme (it does not show how every of the intermediate polynomial sets generated by the execution of the algorithm relate to the *initial* set), therefore a proof attempt using the invariant inference rule will not succeed. Rather, it is a property of the final result of the application of **GB** (again, provided that it terminates).

The property states that for any two elements in the final set that were considered in a reduction diamond:

− either the diamond can be closed (we have a *critical pair*), or

− a new element (built from the result of applying **df** to the two sides of the reduction diamond) is added to the polynomial set (*completion*).

The proof of this property is easy: From Proposition["GB subset"], every intermediary polynomial set generated during the execution of **GB** is a subset of the final result. Therefore, two elements from the final set must be contained in one of the intermediary sets. Then immediately, by a simple application of the definition of the recursive call in Algorithm["CPC scheme"], the proposition is verified.

We will use Proposition["preprocessed CPC scheme"] in the exploration of the first subproblem, **Is−Gröbner-Basis**. The invariant inference rule for **GB** will be used to solve the second subproblem, **ideal equality**. The results are valid only if **GB** terminates.

4.3 Subproblem: Is-Gröbner-Basis

4.3.1 Statement of the Problem

Theorem["Groebner basis specification:2", any[F],
  is-Groebner-basis[GB[F]]]

4.3.2 Knowledge Base

Out of all possible statements on the basic notions of Gröbner bases theory, we give here those which are sufficient for the proof of the above correctness theorem. The proofs of the propositions we give here are not part of this case study. We consider that they are available from earlier exploration rounds.

The following definition links the properties of being a Gröbner basis to the reduction relation modulo polynomial sets:

Definition["is Groebner Basis", any[F],
  is-Groebner-basis[F] $\Leftrightarrow$ is-Church-Rosser[$\longrightarrow_F$]]

The following proposition is Newman's lemma, for power products, from [Buchberger, 2004b]:

Proposition["Newman Lemma:pp", any[G],

$$\text{is-Church-Rosser}[\longrightarrow_G] \Leftrightarrow \forall_p \forall_{f1,f2} \left(\left(\bigwedge \begin{Bmatrix} \text{is-pp}[p] \\ p \longrightarrow_G f1 \\ p \longrightarrow_G f2 \end{Bmatrix} \Rightarrow f1 \downarrow_G f2 \right) \right]$$

We now introduce some properties of the reduction relations:

Proposition["one reduction step:pp", any[is-pp[p], G, f],

$$(p \longrightarrow_G f) \Rightarrow \exists_g \left(\bigwedge \begin{Bmatrix} g \in G \\ \text{lp}[g] \mid p \\ f = \text{rd}[p, g] \end{Bmatrix} \right]$$

Proposition["totally reduces modulo a set", any[f, g, G],
  (f $\twoheadrightarrow_G$ g) $\Longleftarrow$ (g = trd[f, G])]

Proposition["common successor", any[f1, f2, G],

$$f1 \downarrow_G f2 \Leftrightarrow \exists_g \left(\bigwedge \begin{Bmatrix} f1 \twoheadrightarrow_G g \\ f2 \twoheadrightarrow_G g \end{Bmatrix} \right]$$

The following are propositions on the reduction functions, rd, trd, and their connections to the reduction relations:

Proposition["reductions and multiplication", any[a, q, p, g, G],
  trd[rd[a * q * p, g], G] = a * q * trd[rd[p, g], G]]

```
Proposition["diamonds shift", any[g1, g2, G],
  ((g1 ∈ G) ∧ (g2 ∈ G)) ⇒ ∀_{p, a,q} ((trd[rd[p, g1], G] = trd[rd[p, g2], G]) ⇒
      (trd[rd[a ∗ q ∗ p, g1], G] = trd[rd[a ∗ q ∗ p, g2], G]))]
```

```
Proposition["common successor total reduction", any[f1, f2, G],
  (f1 ↓_G f2) ⟸ (trd[f1, G] ↓_G trd[f2, G])]
```

```
Proposition["common successor total reduction", any[f1, f2, G],
  (trd[f1, G] ↓_G trd[f2, G]) ⇒ (f1 ↓_G f2)]
```

The preprocessed algorithm scheme property derived earlier:

```
Proposition["preprocessed CPC scheme", any[g1, g2, F],
  (g1 ∈ GB[F] ∧ g2 ∈ GB[F]) ⇒
    ⋁ { trd[rd[lc[g1, g2], g1], GB[F]] = trd[rd[lc[g1, g2], g2], GB[F]]
        df[trd[rd[lc[g1, g2], g1], GB[F]], trd[rd[lc[g1, g2], g2], GB[F]]] ∈ GB[F] }]
```

Other properties needed in the proofs:

```
Proposition["basic facts: multiplication", any[a, q, p1, p2],
  (p1 = p2) ⇒ (a ∗ q ∗ p1 = a ∗ q ∗ p2)]
```

All the above propositions are collected with the Theory construct in *Theorema*:

```
Theory["GB knowledge",

    Proposition["lc divisibility specification"]
      Definition["is Groebner Basis"]
    Proposition["Newman Lemma:pp"]
    Proposition["one reduction step:pp"]
  Proposition["totally reduces modulo a set"]
    Proposition["common successor"]
Proposition["common successor total reduction"]
    Proposition["preprocessed CPC scheme"]
  Proposition["reductions and multiplication"]
      Proposition["diamonds shift"]
    Proposition["basic facts: multiplication"]
]
```

We now apply the lazy thinking method (and implementation) to synthesize specifications for the unknown algorithms **lc** and **df**.

4.3.3 Lazy Thinking Exploration: Step by Step

First we let the user (this author) organize the exploration process, i.e. let the user play the role of the lazy thinking cascade. The same results that are obtained by letting the user organize the exploration, can be obtained automatically by the exploration cascade (**CascadeLT**), as we will show in the next subsection. However, this manual exploration illustrates the use of the **Failure−Analyzer** and **ConjectureGenerator** implementations, which, as described in Chapter 3, can also be used directly by the user. The focus, in this subsection, is to show how different choices in analysis strategies influence the conjectures that are being generated.

We set up the correctness proof, by calling the *Theorema* Prove command directly.

```
Prove[Theorem["Groebner basis specification:2"], using → Theory["GB knowledge"],
    by → BasicProver,
    ProverOptions →
      {GRWTarget → {"goal", "kb"}, DisableMatchExist → True,
        UseSkolemFunctions → False, RWInsideQuantifiers → True, EarlyCaseDistinction → False,
        DeleteGroundKBFacts → False, UseEqualitiesFirst → False, RWExistentialGoal → True,
        AllowIntroduceQuantifiers → True, ModusPonensUnknownSymbols → {lc, df}}] // Last
```

failed

As expected, the proof fails.

Store the generated proof object for later analysis:

```
groebnerSpecification2Proof := $TmaProofObject
```

Analysis strategy: no universally quantified formulae in the temporary knowledge

Set the option that will not allow universally quantified formulae in the conjecture skeleton:

```
$TmaFAUseUnivQuantif = False
```

False

Now analyze the failing proof by calling FailureAnalyzer:

```
FailureAnalyzer[groebnerSpecification2Proof]
```

{{•lf[24, trd[rd[p_0 , g_0], GB[F_0]] = trd[rd[p_0 , g_1], GB[F_0]], •finfo[]],
•asml[•lf[10.1, g_0 ∈ GB[F_0], •finfo[]], •lf[10.2, lp[g_0] | p_0 , •finfo[]],
•lf[10.3, $f1_0$ = rd[p_0 , g_0], •finfo[]], •lf[14.1, g_1 ∈ GB[F_0], •finfo[]],
•lf[14.2, lp[g_1] | p_0 , •finfo[]], •lf[14.3, $f2_0$ = rd[p_0 , g_1], •finfo[]],
•lf[15.1, trd[rd[lc[g_0 , g_1], g_0], GB[F_0]] = trd[rd[lc[g_0 , g_1], g_1], GB[F_0]], •finfo[]],
•lf[6.1, is-pp[p_0], •finfo[]], •lf[6.2, $p_0 \longrightarrow_{GB[F_0]} f1_0$, •finfo[]], •lf[6.3, $p_0 \longrightarrow_{GB[F_0]} f2_0$, •finfo[]]]}}

The output of the failure analysis above is the conjecture skeleton: a list of the failing goals and the temporary assumptions. Since we did not consider the use of universally quantified formu−lae in the temporary knowledge, a call to the conjecture generation algorithm yields the follow−ing result:

```
GenerateConjectures[groebnerSpecification2Proof, {}, {lc, df}, {}]
```

```
•lma[specification 1, •range[], True,
  •flist[•lf[1, ∀_{F01,g03,g04,p03} ((lp[g03] | p03) ∧ (lp[g04] | p03) ∧ is-pp[p03] ∧ g03 ∈ GB[F01] ∧ g04 ∈
      GB[F01] ∧ (trd[rd[lc[g03, g04], g03], GB[F01]] = trd[rd[lc[g03, g04], g04], GB[F01]]) ⇒
    (trd[rd[p03, g03], GB[F01]] = trd[rd[p03, g04], GB[F01]]))]]]
```

i.e., (after some variable renaming) the *Theorema* internal form of:

```
Lemma["specification 1",
  ∀_{F, g1, g2, p} ((lp[g1] | p) ∧ (lp[g2] | p) ∧ is-pp[p] ∧ g1 ∈ GB[F] ∧ g2 ∈ GB[F] ∧
      (trd[rd[lc[g1, g2], g1], GB[F]] = trd[rd[lc[g1, g2], g2], GB[F]]) ⇒
    (trd[rd[p, g1], GB[F]] = trd[rd[p, g2], GB[F]]))]
```

In the conjecture above, the symbol **GB** is not eliminated, therefore the conjecture generated is a coupled specification for **GB**, **lc**. Remember that the objective of the lazy thinking method is to obtain specifications for the auxiliary subalgorithms in the scheme. The conjecture above does not bring any essentially new information. It states that the theorem is true for the case when the total reduction of the **lc** term modulo the respective arguments totally reduce to the same term. Adding it to the knowledge base would lead to the proof (attempt) getting over the failure.

The situation is not ideal for the exploration, the conjecture produced by the system is not as simple as one could hope. Nonetheless, can it be used in the exploration? That is, can this specification be refined?

We now attempt to do so. Applying the lazy thinking method on this specification, try to prove it, using the knowledge available, Theory["GB knowledge"]:

```
Prove[Lemma["specification 1"], using → Theory["GB knowledge"],
    by → BasicProver,
    ProverOptions →
      {GRWTarget → {"kb", "goal"}, DisableMatchExist → True, UseSkolemFunctions → False,
        RWInsideQuantifiers → True, RWHigherOrder → True, EarlyCaseDistinction → False,
        DeleteGroundKBFacts → False, UseEqualitiesFirst → False, RWExistentialGoal → True,
        AllowIntroduceQuantifiers → True, ModusPonensUnknownSymbols → {lc, df}}] // Last
failed
```

The proof fails. Store the generated proof object for later analysis:

```
specification1Proof := $TmaProofObject
```

We now analyze the proof attempt and generate a conjecture:

```
FailureAnalyzer[specification1Proof]
```

```
{{•lf[3, trd[rd[p0 , g1_0 ], GB[F0 ]] = trd[rd[p0 , g2_0 ], GB[F0 ]], •finfo[]],
    •asml[•lf[2.1, lp[g1_0 ] | p0 , •finfo[]], •lf[2.2, lp[g2_0 ] | p0 , •finfo[]],
      •lf[2.3, is-pp[p0 ], •finfo[]], •lf[2.4, g1_0 ∈ GB[F0 ], •finfo[]], •lf[2.5, g2_0 ∈ GB[F0 ], •finfo[]],
      •lf[2.6, trd[rd[lc[g1_0 , g2_0 ], g1_0 ], GB[F0 ]] = trd[rd[lc[g1_0 , g2_0 ], g2_0 ], GB[F0 ]], •finfo[]]]}}
```

```
GenerateConjectures[specification1Proof, {}, {lc, df}, {}]
```

```
•lma[specification 2, •range[], True,
  •flist[•lf[1, ∀_{F02,g102,g202,p04} ((lp[g102] | p04) ∧ (lp[g202] | p04) ∧ is-pp[p04] ∧
        g102 ∈ GB[F02] ∧ g202 ∈ GB[F02] ∧
        (trd[rd[lc[g102, g202], g102], GB[F02]] = trd[rd[lc[g102, g202], g202], GB[F02]]) ⇒
      (trd[rd[p04, g102], GB[F02]] = trd[rd[p04, g202], GB[F02]]))]]]
```

Unfortunately, the specification generated following the failure analysis is the same as the goal, so no progress was made. Intuitively this says that to prove the goal we need to know the goal.

However, by now allowing universally quantified formulae in the temporary knowledge:

```
$TmaFAUseUnivQuantif = True
```

```
True
```

we get:

```
FailureAnalyzer[specification1Proof]
```

```
{{•lf[3, trd[rd[p0 , g1_0 ], GB[F0 ]] = trd[rd[p0 , g2_0 ], GB[F0 ]], •finfo[]],
   •asml[•lf[1.1, ∀_{g1,g2} (lp[g1] | lc[g1, g2]), •finfo[, GB knowledge]],
     •lf[1.2, ∀_{g1,g2} (lp[g2] | lc[g1, g2]), •finfo[, GB knowledge]],
     •lf[2.1, lp[g1_0 ] | p0 , •finfo[]], •lf[2.2, lp[g2_0 ] | p0 , •finfo[]], •lf[2.3, is-pp[p0 ], •finfo[]],
     •lf[2.4, g1_0 ∈ GB[F0 ], •finfo[]], •lf[2.5, g2_0 ∈ GB[F0 ], •finfo[]],
     •lf[2.6, trd[rd[lc[g1_0 , g2_0 ], g1_0 ], GB[F0 ]] = trd[rd[lc[g1_0 , g2_0 ], g2_0 ], GB[F0 ]], •finfo[]],
     •lf[6, ∀_{a,q} (trd[rd[a ∗ q ∗ lc[g1_0 , g2_0 ], g1_0 ], GB[F0 ]] = trd[rd[a ∗ q ∗ lc[g1_0 , g2_0 ], g2_0 ], GB[F0 ]]),
       •finfo[]]]}}
```

In the above, the highlighted formulae are used to generate the conjecture. The formula in the goal is matched against the universally quantified formula highlighted in the temporary knowl−edge, to give the existential part of the conjecture.

Now generate the corresponding conjecture:

```
GenerateConjectures[specification1Proof, {}, {lc, df}, {}]
```

```
•lma[specification 3, •range[], True, •flist[•lf[1,
      ∀_{g102,g202,p05} ((lp[g102] | p05) ∧ (lp[g202] | p05) ∧ is-pp[p05] ⇒ ∃_{a,q} (p05 = a ∗ q ∗ lc[g102, g202]))]]]
```

The above conjecture is a specification for the **lc** subalgorithm. This is now a decisive moment for the synthesis case study. We delay the discussion on the significance of this conjecture to Subsection 4.3.6.

To summarize, in a step by step lazy thinking exploration (i.e. with the user playing the role of the lazy thinking cascade) after the first unsuccessful proof attempt we applied the failure analysis algorithm, with the default skeleton construction strategy (no universal quantification allowed in the temporary knowledge). The resulting conjecture was then further analyzed: try to

prove the conjecture, analyze the failing proof and the generate of a new conjecture. This analysis showed that the analysis strategies employed do not yield new conjectures. However, allowing universally quantified formulae in the temporary knowledge base leads to a "good" specification, i.e. a specification on the subalgorithm **lc**.

Analysis strategy: universally quantified formulae in the temporary knowledge allowed

Alternatively, if universally quantified formulae are allowed in the in the temporary knowledge from the beginning, in the analysis of the proof of Theorem["Groebner basis specification:2"]:

FailureAnalyzer[groebnerSpecification2Proof]

{{•lf[24, trd[rd[p_0 , g_0], GB[F_0]] = trd[rd[p_0 , g_1], GB[F_0]], •finfo[]],
•asml[•lf[10.1, $g_0 \in$ GB[F_0], •finfo[]], •lf[10.2, lp[g_0] | p_0 , •finfo[]],
•lf[10.3, $f1_0$ = rd[p_0 , g_0], •finfo[]], •lf[1.1, $\forall_{g1,g2}$ (lp[***g1***] | lc[***g1***, ***g2***]), •finfo[, GB knowledge]],
•lf[11, $\forall_{a,q}$ (***a*** ∗ ***q*** ∗ $f1_0$ = ***a*** ∗ ***q*** ∗ rd[p_0 , g_0]), •finfo[]],
•lf[1.2, $\forall_{g1,g2}$ (lp[***g2***] | lc[***g1***, ***g2***]), •finfo[, GB knowledge]], •lf[14.1, $g_1 \in$ GB[F_0], •finfo[]],
•lf[14.2, lp[g_1] | p_0 , •finfo[]], •lf[14.3, $f2_0$ = rd[p_0 , g_1], •finfo[]],
•lf[15.1, trd[rd[lc[g_0 , g_1], g_0], GB[F_0]] = trd[rd[lc[g_0 , g_1], g_1], GB[F_0]], •finfo[]],
•lf[17, $\forall_{a,q}$ (trd[rd[***a*** ∗ ***q*** ∗ lc[g_0 , g_1], g_0], GB[F_0]] = trd[rd[***a*** ∗ ***q*** ∗ lc[g_0 , g_1], g_1], GB[F_0]]), •finfo[]],
•lf[18, $\forall_{a,q}$ (***a*** ∗ ***q*** ∗ trd[rd[lc[g_0 , g_1], g_0], GB[F_0]] = ***a*** ∗ ***q*** ∗ trd[rd[lc[g_0 , g_1], g_1], GB[F_0]]), •finfo[]],
•lf[19, $\forall_{a,q}$ (***a*** ∗ ***q*** ∗ $f2_0$ = ***a*** ∗ ***q*** ∗ rd[p_0 , g_1]), •finfo[]], •lf[6.1, is-pp[p_0], •finfo[]],
•lf[6.2, $p_0 \longrightarrow_{GB[F_0]} f1_0$, •finfo[]], •lf[6.3, $p_0 \longrightarrow_{GB[F_0]} f2_0$, •finfo[]]}}

As before, the highlighted formulae are used to generate the conjecture. The formula in the goal is matched against the universally quantified formula highlighted in the temporary knowledge, to give the existential part of the conjecture.

GenerateConjectures[groebnerSpecification2Proof, {}, {lc, df}, {}]

•lma[specification 3, •range[], True,
•flist[•lf[1, $\forall_{g07,g08,p05}$ ((lp[***g07***] | ***p05***) ∧ (lp[***g08***] | ***p05***) ∧ is-pp[***p05***] ⇒ $\exists_{a,q}$ (***p05*** = ***a*** ∗ ***q*** ∗ lc[***g07***, ***g08***]))]]]

See Subsection 4.3.6 for the discussion on the interpretation of this conjecture. Note however that the conjecture is a specification for the **lc** subalgorithm − and not surprisingly − the same conjecture as generated in the first approach.

Add conjecture to knowledge and continue the exploration

Figure 4.2 illustrates how the analysis strategy influences the manual exploration of the problem (as described above).

The first approach was to start with the default skeleton construction strategy (no universally quantified formulae allowed in the skeleton). This leads to a dead end (the conjecture generator

gives the same conjecture we try to prove). Then we allow universally quantified formulae in the conjecture skeleton, which leads to a new conjecture (on **lc**).

The same conjecture can be obtained directly if we allow universally quantified formulae in the skeleton from the beginning, as we did in the second approach.

Figure 4.2

The conjecture generated is internal representation (after some variable renaming) of :

$$\text{Lemma}\Big[\text{"specification 3"},\ \forall_{g1,\,g2,\,p}\Big((\text{lp}[g1]\mid p)\wedge(\text{lp}[g2]\mid p)\wedge \text{is-pp}[p]\Rightarrow \exists_{a,q}\,(p = a*q*\text{lc}[g1,\,g2])\Big)\Big]$$

As we will discuss in more detail in Subsection 4.3.6 obtaining this conjecture is a *crucial moment* for the exploration of this case study.

The problem is not yet solved (we do not have yet a correctness proof for Theorem["Groebner basis specification:2"]).

We add the conjecture we obtained in the first exploration round to the knowledge base:

```
Theory["GB knowledge:updated",
  Theory["GB knowledge"]
  Lemma["specification 3"]
]
```

and the proof is attempted with this updated knowledge base:

```
Prove[Theorem["Groebner basis specification:2"],
    using → Theory["GB knowledge:updated"],
     by → BasicProver,
    ProverOptions →
      {GRWTarget → {"goal", "kb"}, DisableMatchExist → True,
        UseSkolemFunctions → False, RWInsideQuantifiers → True, EarlyCaseDistinction → False,
        DeleteGroundKBFacts → False, UseEqualitiesFirst → False, RWExistentialGoal → True,
        AllowIntroduceQuantifiers → True, ModusPonensUnknownSymbols → {lc, df}}] // Last
```

failed

The proof fails and we store the proof object for further analysis:

```
groebnerSpecification2Proof2 := $TmaProofObject
```

Now we analyze the proof:

```
FailureAnalyzer[groebnerSpecification2Proof2]
```

$\{\{$•lf[48, trd[rd[lc[g_0 , g_1], g_0], GB[F_0]] = trd[rd[lc[g_0 , g_1], g_1], GB[F_0]], •finfo[]], •asml[
•lf[10.1, g_0 ∈ GB[F_0], •finfo[]], •lf[10.2, lp[g_0] | p_0 , •finfo[]], •lf[10.3, $f1_0$ = rd[p_0 , g_0], •finfo[]],
•lf[1.1, $\forall_{g1,g2}$ (lp[$g1$] | lc[$g1$, $g2$]), •finfo[, GB knowledge:updated, GB knowledge]],
•lf[11, $\forall_{a,q}$ ($a * q * f1_0 = a * q *$ rd[p_0 , g_0]), •finfo[]],
•lf[1.2, $\forall_{g1,g2}$ (lp[$g2$] | lc[$g1$, $g2$]), •finfo[, GB knowledge:updated, GB knowledge]],
•lf[14.1, g_1 ∈ GB[F_0], •finfo[]], •lf[14.2, lp[g_1] | p_0 , •finfo[]],
•lf[14.3, $f2_0$ = rd[p_0 , g_1], •finfo[]], •lf[16, $p_0 = a_0 * q_0 *$ lc[g_0 , g_1], •finfo[]],
•lf[17.2, df[trd[rd[lc[g_0 , g_1], g_0], GB[F_0]], trd[rd[lc[g_0 , g_1], g_1], GB[F_0]]] ∈ GB[F_0], •finfo[]],
•lf[31, $\forall_{a,q}$ ($a * q * p_0 = a * q *$ ($a_0 * q_0 *$ lc[g_0 , g_1])), •finfo[]],
•lf[32, $\forall_{a,q}$ ($a * q * f2_0 = a * q *$ rd[p_0 , g_1]), •finfo[]], •lf[6.1, is-pp[p_0], •finfo[]],
•lf[6.2, $p_0 \longrightarrow_{GB[F_0]} f1_0$, •finfo[]], •lf[6.3, $p_0 \longrightarrow_{GB[F_0]} f2_0$, •finfo[]]]$\}\}$

and generate a conjecture:

```
GenerateConjectures[groebnerSpecification2Proof2, {GB}, {lc, df}, {}]
```

•lma[specification 4, •range[], True,
•flist[•lf[1, $\forall_{F04,g09,g10}$ (df[trd[rd[lc[$g09$, $g10$], $g09$], GB[$F04$]], trd[rd[lc[$g09$, $g10$], $g10$], GB[$F04$]]] ∈
GB[$F04$] ∧ $g09$ ∈ GB[$F04$] ∧ $g10$ ∈ GB[$F04$] ⇒
(trd[rd[lc[$g09$, $g10$], $g09$], GB[$F04$]] = trd[rd[lc[$g09$, $g10$], $g10$], GB[$F04$]]))]]]

The conjecture generated is a coupled conjecture on **lc, df, GB**. We allowed universally quanti−fied formulae in the conjecture skeleton, but none of these matched the goal. The discussion on the interpretation of this conjecture is deferred for Subsection 4.3.6.

The output of the conjecture generator is the *Theorema* internal form (after some variable renaming) of:

```
Lemma["specification 4",
  ∀F,g1,g2 (df[trd[rd[lc[g1, g2], g1], GB[F]], trd[rd[lc[g1, g2], g2], GB[F]]] ∈ GB[F] ∧
      g1 ∈ GB[F] ∧ g2 ∈ GB[F] ⇒
    (trd[rd[lc[g1, g2], g1], GB[F]] = trd[rd[lc[g1, g2], g2], GB[F]]))]
```

At this point, we continue the exploration, by adding the conjecture to the knowledge base:

```
Theory["GB knowledge:updated:2",
  Theory["GB knowledge:updated"]
    Lemma["specification 4"]]
```

and calling:

```
Prove[Theorem["Groebner basis specification:2"],
  using → Theory["GB knowledge:updated:2"],
  by → BasicProver,
  ProverOptions →
    {GRWTarget → {"goal", "kb"}, DisableMatchExist → True,
      UseSkolemFunctions → False, RWInsideQuantifiers → True, EarlyCaseDistinction → False,
      DeleteGroundKBFacts → False, UseEqualitiesFirst → False, RWExistentialGoal → True,
      AllowIntroduceQuantifiers → True, ModusPonensUnknownSymbols → {lc, df}}] // Last
proved
```

The proof succeeds (*Theorema* proves it), and our exploration process ends.

To summarize, we set up the correctness proof of Theorem["Groebner basis specification:2"]. The proof failed, but using our failure analysis and conjecture generator implementation (**FailureAnalyzer, GenerateConjectures**) added new conjectures to the knowledge base, which led to a successful proof.

The user (this author) played the role of the cascade mechanism in organizing the exploration process. But **CascadeLT**, our implementation of the lazy thinking cascade can also be used to do the exploration, with the same results.

4.3.4 Lazy Thinking Exploration: Automated

Now let **CascadeLT** organize the exploration process:

```
Prove[Theorem["Groebner basis specification:2"], using → Theory["GB knowledge"],
  by → CascadeLT[BasicProver, GenerateConjectures, {}, {lc, df}, •asml[], •asml[]],
  ProverOptions →
    {GRWTarget → {"goal", "kb"}, DisableMatchExist → True, UseSkolemFunctions → False,
      RWInsideQuantifiers → True, EarlyCaseDistinction → False,
      DeleteGroundKBFacts → False, UseEqualitiesFirst → False, RWExistentialGoal → True,
      AllowIntroduceQuantifiers → True, ModusPonensUnknownSymbols → {lc, df}}]
```

LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•If[Lemma (specification 2): 1,

$$\forall_{g05,g06,p04}\left((\text{lp}[g05]\mid p04)\wedge(\text{lp}[g06]\mid p04)\wedge\text{is-pp}[p04]\Rightarrow\exists_{a,q}\,(p04=a*q*\text{lc}[g05,g06])\right),\ \bullet\text{finfo}[]\Big]$$

Now attempt the proof with the updated knowledge base.

LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•If[Lemma (specification 4): 1,

$$\forall_{F04,g09,g10}\,(\text{df}[\text{trd}[\text{rd}[\text{lc}[g09,g10],g09],\text{GB}[F04]],\text{trd}[\text{rd}[\text{lc}[g09,g10],g10],\text{GB}[F04]]]\in\text{GB}[F04]\wedge$$
$$g09\in\text{GB}[F04]\wedge g10\in\text{GB}[F04]\Rightarrow$$
$$(\text{trd}[\text{rd}[\text{lc}[g09,g10],g09],\text{GB}[F04]]=\text{trd}[\text{rd}[\text{lc}[g09,g10],g10],\text{GB}[F04]])),\ \bullet\text{finfo}[]\Big]$$

Now attempt the proof with the updated knowledge base.

LAZY THINKING ::::: The proof is completed!

The exploration process is successful. Note that the conjectures generated during this process are the same as in the manual exploration in the previous Subsection 4.3.2.

4.3.5 *Theorema* Proof

We now give the final proof in the exploration, produced by *Theorema*. It makes use of the conjectures generated during the exploration rounds and automatically added by the lazy think−ing implementation during its execution.

In this proof, we highlight: the conjectures added during the exploration process, and the place in the proof where these are used.

Prove:

(Theorem (Groebner basis specification:2)) $\forall_F$ is-Groebner-basis[GB[F]],

under the assumptions:

(Proposition (lc divisibility specification)) $\forall_{g1,g2}\,((\text{lp}[g1]\mid\text{lc}[g1,g2])\wedge(\text{lp}[g2]\mid\text{lc}[g1,g2]))$,

(Definition (is Groebner Basis)) $\forall_F\,(\text{is-Groebner-basis}[F]\Leftrightarrow\text{is-Church-Rosser}[\longrightarrow_F])$,

(Proposition (Newman Lemma:pp))

$$\forall_G\left(\text{is-Church-Rosser}[\longrightarrow_G]\Leftrightarrow\forall_p\,\forall_{f1,f2}\,(\text{is-pp}[p]\wedge p\longrightarrow_G f1\wedge p\longrightarrow_G f2\Rightarrow f1\downarrow_G f2)\right),$$

(Proposition (one reduction step:pp))

$$\forall_{\substack{\text{is-pp}[p]\\ G\\ f}} \left(p \longrightarrow_G f \Rightarrow \exists_g (g \in G \wedge (\text{lp}[g] \mid p) \wedge (f = \text{rd}[p, g])) \right),$$

(Proposition (totally reduces modulo a set)) $\forall_{f,g,G} ((f \twoheadrightarrow_G g) \Leftarrow (g = \text{trd}[f, G]))$,

(Proposition (common successor)) $\forall_{f1,f2,G} \left(f1 \downarrow_G f2 \Leftrightarrow \exists_g (f1 \twoheadrightarrow_G g \wedge f2 \twoheadrightarrow_G g) \right)$,

(Proposition (common successor total reduction))

$\forall_{f1,f2,G} (\text{trd}[f1, G] \downarrow_G \text{trd}[f2, G] \Rightarrow f1 \downarrow_G f2)$,

(Proposition (preprocessed CPC scheme))

$$\forall_{g1,g2,F} (g1 \in \text{GB}[F] \wedge g2 \in \text{GB}[F] \Rightarrow \quad ,$$
$$(\text{trd}[\text{rd}[\text{lc}[g1, g2], g1], \text{GB}[F]] = \text{trd}[\text{rd}[\text{lc}[g1, g2], g2], \text{GB}[F]]) \vee$$
$$\text{df}[\text{trd}[\text{rd}[\text{lc}[g1, g2], g1], \text{GB}[F]], \text{trd}[\text{rd}[\text{lc}[g1, g2], g2], \text{GB}[F]]] \in \text{GB}[F])$$

(Proposition (reductions and multiplication))

$\forall_{a,q,p,g,G} (\text{trd}[\text{rd}[a * q * p, g], G] = a * q * \text{trd}[\text{rd}[p, g], G])$,

(Proposition (diamonds shift))

$$\forall_{g1,g2,G} \left(g1 \in G \wedge g2 \in G \Rightarrow \forall_{p,a,q} ((\text{trd}[\text{rd}[p, g1], G] = \text{trd}[\text{rd}[p, g2], G]) \Rightarrow, \right.$$
$$\left. (\text{trd}[\text{rd}[a * q * p, g1], G] = \text{trd}[\text{rd}[a * q * p, g2], G])) \right)$$

(Proposition (basic facts: multiplication)) $\forall_{a,q,p1,p2} ((p1 = p2) \Rightarrow (a * q * p1 = a * q * p2))$,

(Lemma (specification 2): 1)

$$\forall_{g05,g06,p04} \left((\text{lp}[g05] \mid p04) \wedge (\text{lp}[g06] \mid p04) \wedge \text{is-pp}[p04] \Rightarrow \exists_{a,q} (p04 = a * q * \text{lc}[g05, g06]) \right),$$

(Lemma (specification 4): 1)

$$\forall_{F04,g09,g10} (df[trd[rd[lc[g09, g10], g09], GB[F04]], trd[rd[lc[g09, g10], g10], GB[F04]]] \in GB[F04] \wedge g09 \in GB[F04] \wedge g10 \in GB[F04] \Rightarrow (trd[rd[lc[g09, g10], g09], GB[F04]] = trd[rd[lc[g09, g10], g10], GB[F04]]))$$

Proof.

Formula (Proposition (lc divisibility specification)) is simplified to:

(1) $\forall_{g1,g2} (lp[g1] \mid lc[g1, g2]) \bigwedge \forall_{g1,g2} (lp[g2] \mid lc[g1, g2])$.

For proving (Theorem (Groebner basis specification:2)) we take all variables arbitrary but fixed and prove:

(2) is-Groebner-basis[GB[F_0]].

Formula (2), using (Definition (is Groebner Basis)), is implied by:

(3) is-Church-Rosser[$\longrightarrow_{GB[F_0]}$].

Formula (3), using (Proposition (Newman Lemma:pp)), is implied by:

(4) $\forall_{p} \forall_{f1,f2} (is\text{-}pp[p] \wedge p \longrightarrow_{GB[F_0]} f1 \wedge p \longrightarrow_{GB[F_0]} f2 \Rightarrow f1 \downarrow_{GB[F_0]} f2)$.

For proving (4) we take all variables arbitrary but fixed and prove:

(5) $\forall_{f1,f2} (is\text{-}pp[p_0] \wedge p_0 \longrightarrow_{GB[F_0]} f1 \wedge p_0 \longrightarrow_{GB[F_0]} f2 \Rightarrow f1 \downarrow_{GB[F_0]} f2)$.

We assume

(6) $is\text{-}pp[p_0] \wedge p_0 \longrightarrow_{GB[F_0]} f1_0 \wedge p_0 \longrightarrow_{GB[F_0]} f2_0$,

and show

(7) $f1_0 \downarrow_{GB[F_0]} f2_0$.

By modus ponens, from (6.2) and an appropriate instance of (Proposition (one reduction step:pp)), with the substitution $f := f1_0$, $G :=$ GB[F_0], and $p := p_0$, we need to show :

(9) is-pp[p_0]

and then we can add to the knowledge base:

(8) $\underset{\boldsymbol{g}}{\exists}\,(\boldsymbol{g} \in \mathrm{GB}[F_0] \wedge (\mathrm{lp}[\boldsymbol{g}] \mid p_0) \wedge (f1_0 = \mathrm{rd}[p_0, \boldsymbol{g}]))$,

Formula (9) is true because it is identical to (6.1).

Now the application of the modus ponens is correct and we go on with the proof.

By (8) we can take appropriate values such that:

(10) $g_0 \in \mathrm{GB}[F_0] \wedge (\mathrm{lp}[g_0] \mid p_0) \wedge (f1_0 = \mathrm{rd}[p_0, g_0])$.

By modus ponens, from (10.3) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(11) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall}\,(\boldsymbol{a} * \boldsymbol{q} * f1_0 = \boldsymbol{a} * \boldsymbol{q} * \mathrm{rd}[p_0, g_0])$,

By modus ponens, from (6.3) and an appropriate instance of (Proposition (one reduction step:pp)), with the substitution $\boldsymbol{f} := f2_0$, $\boldsymbol{G} := \mathrm{GB}[F_0]$, and $\boldsymbol{p} := p_0$, we need to show :

(13) is-pp$[p_0]$

and then we can add to the knowledge base:

(12) $\underset{\boldsymbol{g}}{\exists}\,(\boldsymbol{g} \in \mathrm{GB}[F_0] \wedge (\mathrm{lp}[\boldsymbol{g}] \mid p_0) \wedge (f2_0 = \mathrm{rd}[p_0, \boldsymbol{g}]))$,

Formula (13) is true because it is identical to (6.1).

Now the application of the modus ponens is correct and we go on with the proof.

By (12) we can take appropriate values such that:

(14) $g_1 \in \mathrm{GB}[F_0] \wedge (\mathrm{lp}[g_1] \mid p_0) \wedge (f2_0 = \mathrm{rd}[p_0, g_1])$.

By modus ponens, from (14.2), (10.2), (6.1) and an appropriate instance of (Lemma (specification 2): 1) follows:

(15) $\underset{\boldsymbol{a},\boldsymbol{q}}{\exists}\,(p_0 = \boldsymbol{a} * \boldsymbol{q} * \mathrm{lc}[g_0, g_1])$,

By (15) we can take appropriate values such that:

(16) $p_0 = a_0 * q_0 * \mathrm{lc}[g_0, g_1]$.

By modus ponens, from (10.1), (14.1) and an appropriate instance of (Proposition (preprocessed CPC scheme)) follows:

(17) $(\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]) \vee$,
$\mathrm{df}[\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]], \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]] \in \mathrm{GB}[F_0]$

We prove (7) by case distinction using (17).

Case (17.1) $\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]$:

By modus ponens, from (10.1), (14.1) and an appropriate instance of (Proposition (diamonds shift)) follows:

(18) $\underset{\boldsymbol{p},\boldsymbol{a},\boldsymbol{q}}{\forall} ((\mathrm{trd}[\mathrm{rd}[\boldsymbol{p}, g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{p}, g_1], \mathrm{GB}[F_0]]) \Rightarrow$,
$(\mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \boldsymbol{p}, g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \boldsymbol{p}, g_1], \mathrm{GB}[F_0]]))$

By modus ponens, from (17.1) and an appropriate instance of (18) follows:

(19) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall} (\mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]])$,

By modus ponens, from (17.1) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(20) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall} (\boldsymbol{a} * \boldsymbol{q} * \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \boldsymbol{a} * \boldsymbol{q} * \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]])$,

By modus ponens, from (16) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(21) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall} (\boldsymbol{a} * \boldsymbol{q} * p_0 = \boldsymbol{a} * \boldsymbol{q} * (a_0 * q_0 * \mathrm{lc}[g_0, g_1]))$,

By modus ponens, from (14.3) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(22) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall} (\boldsymbol{a} * \boldsymbol{q} * f2_0 = \boldsymbol{a} * \boldsymbol{q} * \mathrm{rd}[p_0, g_1])$,

Formula (7), using (14.3), is implied by:

(23) $f1_0 \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[p_0, g_1]$.

Formula (23), using (16), is implied by:

(24) $f1_0 \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1]$.

Formula (24), using (10.3), is implied by:

(25) $\mathrm{rd}[p_0, g_0] \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1]$.

Formula (25), using (16), is implied by:

(26) $\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_0] \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1]$.

Formula (26), using (Proposition (common successor)), is implied by:

(27) $\exists_{\boldsymbol{g}} (\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_0] \twoheadrightarrow_{\mathrm{GB}[F_0]} \boldsymbol{g} \wedge \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1] \twoheadrightarrow_{\mathrm{GB}[F_0]} \boldsymbol{g})$.

Formula (27), using (Proposition (totally reduces modulo a set)), is implied by:

(28)

$\exists_{\boldsymbol{g}} ((\boldsymbol{g} = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]]) \wedge (\boldsymbol{g} = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]))$.

In order to prove (28), it suffices to prove

(29) $\mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]$.

Formula (29) is proved because it is an instance of (19).

Case (17.2) $\mathrm{df}[\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]], \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]] \in \mathrm{GB}[F_0]$:

By modus ponens, from (17.2), (14.1), (10.1) and an appropriate instance of (Lemma (specification 4): 1) follows:

(30) $\mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]]$,

By modus ponens, from (10.1), (14.1) and an appropriate instance of (Proposition (diamonds shift)) follows:

(31) $\forall_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{q}} ((\mathrm{trd}[\mathrm{rd}[\boldsymbol{p}, g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{p}, g_1], \mathrm{GB}[F_0]]) \Rightarrow$,
$(\mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \boldsymbol{p}, g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \boldsymbol{p}, g_1], \mathrm{GB}[F_0]]))$

By modus ponens, from (30) and an appropriate instance of (31) follows:

(32) $\forall_{\boldsymbol{a},\boldsymbol{q}} (\mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[\boldsymbol{a} * \boldsymbol{q} * \mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]])$,

By modus ponens, from (30) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(33) $\forall_{\boldsymbol{a},\boldsymbol{q}} (\boldsymbol{a} * \boldsymbol{q} * \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_0], \mathrm{GB}[F_0]] = \boldsymbol{a} * \boldsymbol{q} * \mathrm{trd}[\mathrm{rd}[\mathrm{lc}[g_0, g_1], g_1], \mathrm{GB}[F_0]])$,

By modus ponens, from (16) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(34) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall}\ (\boldsymbol{a} * \boldsymbol{q} * p_0 = \boldsymbol{a} * \boldsymbol{q} * (a_0 * q_0 * \mathrm{lc}[g_0 , g_1]))$,

By modus ponens, from (14.3) and an appropriate instance of (Proposition (basic facts: multiplication)) follows:

(35) $\underset{\boldsymbol{a},\boldsymbol{q}}{\forall}\ (\boldsymbol{a} * \boldsymbol{q} * f2_0 = \boldsymbol{a} * \boldsymbol{q} * \mathrm{rd}[p_0 , g_1])$,

Formula (7), using (14.3), is implied by:

(37) $f1_0 \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[p_0 , g_1]$.

Formula (37), using (16), is implied by:

(39) $f1_0 \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1]$.

Formula (39), using (10.3), is implied by:

(41) $\mathrm{rd}[p_0 , g_0] \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1]$.

Formula (41), using (16), is implied by:

(43) $\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_0] \downarrow_{\mathrm{GB}[F_0]} \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1]$.

Formula (43), using (Proposition (common successor)), is implied by:

(45) $\underset{\boldsymbol{g}}{\exists}\ (\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_0] \twoheadrightarrow_{\mathrm{GB}[F_0]} \boldsymbol{g} \wedge \mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1] \twoheadrightarrow_{\mathrm{GB}[F_0]} \boldsymbol{g})$.

Formula (45), using (Proposition (totally reduces modulo a set)), is implied by:

(47)

$\underset{\boldsymbol{g}}{\exists}\ ((\boldsymbol{g} = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_0], \mathrm{GB}[F_0]]) \wedge (\boldsymbol{g} = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1], \mathrm{GB}[F_0]]))$.

In order to prove (47), it suffices to prove

(48) $\mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_0], \mathrm{GB}[F_0]] = \mathrm{trd}[\mathrm{rd}[a_0 * q_0 * \mathrm{lc}[g_0 , g_1], g_1], \mathrm{GB}[F_0]]$.

Formula (48) is proved because it is an instance of (32).

□

4.3.6 Interpretation of the Results

Consider the first conjecture generated by the lazy thinking method (with some variable renaming):

$$\text{Lemma}\Big["\text{(specification 2): 1}",\ \forall_{g1,g2,p}\Big((\text{lp}[g1] \mid p) \wedge (\text{lp}[g2] \mid p) \wedge \text{is-pp}[p] \Rightarrow \exists_{a,q}\ (p = a * q * \text{lc}[g1, g2])\Big)\Big]$$

(This is the same as Lemma["specification 3"] generated in the manual exploration in Subsec−tion 4.3.3.)

Immediately, by the definition of divisibility, we can rewrite this as:

$$\text{Proposition}\Big["\text{(specification 2): 1 modified}",\ \forall_{g1,g2,p}\ ((\text{lp}[g1] \mid p) \wedge (\text{lp}[g2] \mid p) \wedge \text{is-pp}[p] \Rightarrow (\text{lc}[g1, g2] \mid p))\Big]$$

We also have the specification attached to the polynomial CPC scheme:

$$\text{Proposition}\Big["\text{lc divisibility specification}",\ \forall_{g1,g2}\ \bigwedge\begin{Bmatrix}\text{lp}[g1] \mid \text{lc}[g1, g2]\\ \text{lp}[g2] \mid \text{lc}[g1, g2]\end{Bmatrix}\Big]$$

Proposition["(specification 2): 1 modified"] and Proposition["lc divisibility specification"] are specification for **lc**, that say the following:

− if the leading power product of arbitrary polynomials g1, g2 divide the arbitrary power prod−uct p, then this p is a multiple of lc[g1, g2], and

− the leading power product of g1, g2 respectively, divide lc[g1, g2].

But this is none other that the usual definition of the least common multiple:

$$\text{lc}[g1, g2] = \text{lcm}[\text{lp}[g1], \text{lp}[g2]]$$

From [Buchberger, 2004b]: "Eureka! The crucial function **lc** (the "critical pair" function) in the critical pair / completion algorithm scheme has been "automatically" synthesized!" We have shown that our implementation of lazy thinking can indeed synthesize the **lc** algorithm.

Now for the second conjecture generated in the lazy thinking exploration (after some variable renaming):

```
Lemma["(specification 4): 1",
  ∀_{F, g1, g2} (df[trd[rd[lc[g1, g2], g1], GB[F]], trd[rd[lc[g1, g2], g2], GB[F]]] ∈ GB[F] ∧
      g1 ∈ GB[F] ∧ g1 ∈ GB[F] ⇒
    (trd[rd[lc[g1, g2], g1], GB[F]] = trd[rd[lc[g1, g2], g2], GB[F]]))]
```

(This is the same as Lemma["specification 3"] generated in the manual exploration in Subsection 4.3.3.)

The **GB** symbol could not be eliminated from the conjecture, which means we have a conjecture on **GB**, **lc** and **df**, which is possibly more difficult to handle than the initial problem. We need to try other approaches to get a conjecture on **df**. Note, however the intuition behind the generated conjecture: if the result of applying the **df** function to its two arguments (totally reduced from **lc**, following 2 distinct paths) stays in the final set, then the problem can be solved.

4.4 Subproblem: Ideal Equality

4.4.1 Statement of the Problem

```
Theorem["Groebner basis specification:3", any[F],
  ideal[GB[F]] = ideal[F]]
```

The (inverse) inclusion:

```
Theorem["Groebner basis specification:3 ⊇", any[F],
  ideal[GB[F]] ⊇ ideal[F]]
```

is easy: polynomials are added to the initial set. This preserves the ideal.

Therefore we consider here only the (direct) implication:

```
Theorem["Groebner basis specification: 3 ⊆", any[F],
  ideal[GB[F]] ⊆ ideal[F]]
```

4.4.2 Knowledge Base

```
Proposition["non strict inclusion:reflexivity", any[A],
  A ⊆ A]
```

Knowledge on ideals:

```
Proposition["ideal inclusion: bigger generator set", any[F, x],
  (ideal[F ⌢ x ] ⊆ ideal[F]) ⇔ (x ∈ ideal[F])]
```

Knowledge on pairs:

```
Proposition["pairs and cartesian product", any[F],
  pairs[F] ⊆ F × F]
```

```
Proposition["completion of pairs and cartesian product", any[F, x, p],
  ⟨p⟩ ⊆ F×F ⇒ ⟨p⟩ ≍ ⟨⟨F_k, x⟩ | k=1,...,|F|⟩ ⊆ (F ⌢ x)×(F ⌢ x)]
```

Collecting all the knowledge:

```
Theory["GB knowledge:ideals",
        Algorithm["CPC scheme"]
    Proposition["non strict inclusion:reflexivity"]
  Proposition["ideal inclusion: bigger generator set"]
      Proposition["pairs and cartesian product"]]
```

4.4.3 Lazy Thinking Exploration

Note that if the property we want to prove holds for intermediate sets, i.e. the ideal generated by the intermediary sets stays in the ideal generated by the initial set, then if/when the algorithm terminates, the final set (the Gröbner basis) will generate the same algorithm. We can therefore use the CPC invariant inference rule introduced earlier.

Proof call:

```
Prove[Theorem["Groebner basis specification: 3 ⊆"],
    using → Theory["GB knowledge:ideals"], by → CPCInduction, ProverOptions →
      {GRWTarget → {"goal", "kb"}}] // Last
failed
```

Store the generated proof object for later analysis:

```
specification3ProofObject := $TmaProofObject
```

The proof fails (as expected): nothing is known about **df**. However, the conjecture generator suggests the following conjecture (specification for **df**):

```
GenerateConjectures[specification3ProofObject, {}, {lc, df}, {}]
```

```
•lma[specification 1, •range[], True,
  •flist[•If[1, ∀_{F01,G101,g301,g401} (g301 ∈ G101 ∧ g401 ∈ G101 ∧ trd[rd[lc[g301, g401], g301], G101] ≠
        trd[rd[lc[g301, g401], g401], G101] ∧ ideal[G101] ⊆ ideal[F01] ⇒
      ideal[G101] ⌢ df[trd[rd[lc[g301, g401], g301], G101], trd[rd[lc[g301, g401], g401], G101]]] ⊆
        ideal[F01])]]]
```

Adding this conjecture to the knowledge base makes the proof succeed, as shown by the call to the lazy thinking implementation:

```
Prove[Theorem["Groebner basis specification: 3 ⊆"],
  using → Theory["GB knowledge:ideals"],
  by → CascadeLT[CPCInduction, GenerateConjectures, {}, {lc, df}, •asml[], •asml[]],
  ProverOptions →
    {GRWTarget → {"goal", "kb"}}]
```

LAZY THINKING::::: The proof fails.
After analysing the failing proof, the following conjecture is added to the knowledge base:
•If[Lemma (specification 2): 1,

$$\forall_{F02,G102,g302,g402} (g302 \in G102 \wedge g402 \in G102 \wedge \text{trd}[\text{rd}[\text{lc}[g302, g402], g302], G102] \neq \text{trd}[\text{rd}[\text{lc}[g302, g402], g402], G102] \wedge \text{ideal}[G102] \subseteq \text{ideal}[F02] \Rightarrow \text{ideal}[G102 \frown \text{df}[\text{trd}[\text{rd}[\text{lc}[g302, g402], g302], G102], \text{trd}[\text{rd}[\text{lc}[g302, g402], g402], G102]]] \subseteq \text{ideal}[F02]),$$

•finfo[]]
Now attempt the proof with the updated knowledge base.

LAZY THINKING ::::: The proof is completed!!!!

4.4.4 *Theorema* Proof

We include the last (successful) proof in the lazy thinking exploration. Again, we highlight the conjecture added during the exploration process and where this is used during the proof.

Prove:

(Theorem (Groebner basis specification: 3 $\subseteq$)) $\forall_F (\text{ideal}[\text{GB}[F]] \subseteq \text{ideal}[F])$,

under the assumptions:

(Algorithm (CPC scheme): 1) $\forall_F (\text{GB}[F] = \text{GB}[F, \text{pairs}[F]])$,

(Algorithm (CPC scheme): 2) $\forall_F (\text{GB}[F, \langle\rangle] = F)$,

(Algorithm (CPC scheme): 3)

$$\forall_{F,g1,g2,\overline{p}} \left(\text{GB}[F, \langle\langle g1, g2\rangle, \overline{p}\rangle] = \left\| \text{GB}[F, \langle \overline{p} \rangle] \Leftarrow h1 = h2, \right. \right.$$

$$\text{GB}\left[F \frown \text{df}[h1, h2], \langle \overline{p} \rangle \asymp \left\langle \langle F_k, \text{df}[h1, h2] \rangle \Big|_{k=1,\ldots,|F|} \right\rangle\right] \Leftarrow \text{otherwise} \left. \right\| \left. \right| [$$

$$f \leftarrow \text{lc}[g1, g2], h1 \leftarrow \text{trd}[\text{rd}[f, g1], F], h2 \leftarrow \text{trd}[\text{rd}[f, g2], F]] \Big)$$

(Proposition (non strict inclusion:reflexivity)) $\forall_A (A \subseteq A)$,

(Proposition (ideal inclusion: bigger generator set))

$$\forall_{F,x} \left(\text{ideal}[F \frown x] \subseteq \text{ideal}[F] \Leftrightarrow x \in \text{ideal}[F]\right),$$

(Proposition (pairs and cartesian product)) $\forall_{F} \left(\text{pairs}[F] \subseteq F \times F\right)$,

(Lemma (specification 2): 1)

$$\forall_{F02,G102,g302,g402} \left(g302 \in G102 \wedge g402 \in G102 \wedge \right.$$
$$\text{trd}[\text{rd}[\text{lc}[g302, g402], g302], G102] \neq \text{trd}[\text{rd}[\text{lc}[g302, g402], g402], G102] \wedge$$
$$\text{ideal}[G102] \subseteq \text{ideal}[F02] \Rightarrow \text{ideal}[G102 \frown \text{df}[\text{trd}[\text{rd}[\text{lc}[g302, g402], g302], G102],$$
$$\left.\text{trd}[\text{rd}[\text{lc}[g302, g402], g402], G102]]] \subseteq \text{ideal}[F02]\right).$$

We prove (Theorem (Groebner basis specification: 3 $\subseteq$)), an invariant property of the CPC algorithm scheme, using its recursive structure.

Proof.

We have to prove,

(2)

$$\forall_{G2,g5,g6} \left(g5 \in G2 \wedge g6 \in G2 \wedge \right.$$
$$\text{trd}[\text{rd}[\text{lc}[g5, g6], g5], G2] \neq \text{trd}[\text{rd}[\text{lc}[g5, g6], g6], G2] \wedge \text{ideal}[G2] \subseteq \text{ideal}[F_0] \Rightarrow$$
$$\left.\text{ideal}[G2 \frown \text{df}[\text{trd}[\text{rd}[\text{lc}[g5, g6], g5], G2], \text{trd}[\text{rd}[\text{lc}[g5, g6], g6], G2]]] \subseteq \text{ideal}[F_0]\right)$$

i.e. the property holds for the case when the reduction diamond cannot be closed.

We assume

(4)

$$g5_0 \in G2_0 \wedge g6_0 \in G2_0 \wedge$$
$$\text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g5_0], G2_0] \neq \text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g6_0], G2_0] \wedge \text{ideal}[G2_0] \subseteq \text{ideal}[F_0],$$

and show

(5)

$$\text{ideal}[G2_0 \frown \text{df}[\text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g5_0], G2_0], \text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g6_0], G2_0]]] \subseteq \text{ideal}[F_0].$$

Formula (5), using (Lemma (specification 2): 1), is implied by:

(6)

$g5_0 \in G2_0 \bigwedge g6_0 \in G2_0 \bigwedge$.
$\text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g5_0], G2_0] \neq \text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g6_0], G2_0] \bigwedge \text{ideal}[G2_0] \subseteq \text{ideal}[F_0]$

We prove the individual conjunctive parts of (6):

Proof of (6.1) $g5_0 \in G2_0$:

Formula (6.1) is true because it is identical to (4.1).

Proof of (6.2) $g6_0 \in G2_0$:

Formula (6.2) is true because it is identical to (4.2).

Proof of (6.3) $\text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g5_0], G2_0] \neq \text{trd}[\text{rd}[\text{lc}[g5_0, g6_0], g6_0], G2_0]$:

Formula (6.3) is true because it is identical to (4.3).

Proof of (6.4) $\text{ideal}[G2_0] \subseteq \text{ideal}[F_0]$:

Formula (6.4) is true because it is identical to (4.4).

□

4.4.5 Interpretation of the Result

The conjecture generated during the exploration process is (with some variable renaming):

```
Lemma["(specification 2): 1",
  ∀_{F, G, g1, g2} (g1 ∈ G ∧ g2 ∈ G ∧
      trd[rd[lc[g1, g2], g1], G] ≠ trd[rd[lc[g1, g2], g2], G] ∧ ideal[G] ⊆ ideal[F] ⇒
    ideal[G ∽ df[trd[rd[lc[g1, g2], g1], G], trd[rd[lc[g1, g2], g2], G]]] ⊆ ideal[F])
]
```

Note that this is an exploration session different from the one described in Section 4.3, and the names of the conjectures are generated automatically. Therefore, the clash on the labels of the conjectures is not a problem.

The conjecture generated essentially says that if,

− we have polynomial sets F, G such that ideal[G] ⊆ ideal[F], and
− polynomials g1, g2 such that

− the diamond obtained by
    − reducing lc[g1, g2] modulo g1, and modulo g2 respectively, then
    − totally reducing modulo G the result to
        h1= trd[rd[lc[g1, g2], g1], G],
        h2=trd[rd[lc[g1, g2], g2], G], respectively,
cannot be closed (h1≠h2),

then

− adding the term df[h1, h2] to the set G has to preserve the ideal inclusion, i.e.
ideal[G⌢df[h1, h2]] ⊆ ideal[F].

Note that in the proof of the proposition little specific knowledge on ideals was used. This knowledge, available from previous exploration rounds can, at this point, be used to identify binary operations that satisfy the specification.

The first place to look are the basic binary operations on polynomials, and it is relatively easy to check that the substraction of polynomials satisfies the specification:

We want to express df[h1, h2] as a (linear) combination of elements that generate the ideal F.

Since
    − h1 is the result of reducing the element lc[g1, g2] modulo g1 then totally reducing the result modulo G, and
    − h2 is the result of reducing the element lc[g1, g2] modulo g2 then totally reducing the result modulo G,

from the definitions of reduction (Definition["reduction modulo polynomials"]) and total reduc−tion (repeated application of reduction),

    − h1 can be written as lc[g1, g2] + (linear) combination of generating elements from F,
    − h2 can be written as lc[g1, g2] + (linear) combination of generating elements from F.

From this, immediately, we see that h1 − h2 yields a (linear) combination of generating ele−ments from F, which clearly places h1−h2 in ideal[F].

We have shown that, by applying some exploration steps (retrieval), we can identify the differ−ence of polynomials as a solution for **df**.

At this point, we have synthesized by lazy thinking exploration, the algorithm for Gröbner bases: The unknown functions **lc** (the critical pair function) and **df** (the completion function) have been found, therefore all the ingredients of Algorithm["CPC scheme"] are known. We have partial correctness.

Note that the two functions we synthesized, **lc** and **df**, are equivalent to the notion of *S−polyno−mials*, see, for example, [Buchberger, 1998].

As soon as **lc** and **df** are known, the termination proof can be formulated.

4.5 Termination

The lazy thinking method does not play a role in proving the termination of the instantiation of the CPC scheme, Algorithm["CPC Scheme"]. However, the inference rule we introduced for proving properties of the algorithm (and therefore all the proofs we carried out) are correct only if the algorithm terminates.

Recall

$$\begin{array}{l}\text{Algorithm}[\text{“CPC scheme”}, \text{any}[F, g1, g2, \overline{p}],\\ \qquad GB[F] = GB[F, pairs[F]]\\ \qquad GB[F, \langle\rangle] = F\\ \qquad GB[F, \langle\langle g_1, g_2\rangle, \overline{p}\rangle] =\\ \qquad\quad \text{where}[f = lc[g_1, g_2], h_1 = trd[rd[f, g_1], F], h_2 = trd[rd[f, g_2], F], \qquad\qquad].\\ \qquad\quad \left\{\begin{array}{lll} GB[F, \langle\overline{p}\rangle] & \Leftarrow & h_1 = h_2\\ GB[F \frown df[h_1, h_2], \langle\overline{p}\rangle \asymp \left\langle \langle F_k, df[h_1, h_2]\rangle \underset{k=1,\ldots,|F|}{|} \right\rangle] & \Leftarrow & \text{otherwise}\end{array}\right.]\end{array}$$

Termination depends on the termination of the binary function GB. Showing termination amounts to showing that we can find a well-founded ordering on the arguments in the recursive call. For binary function, we look for a lexicographic ordering: we first compare the first argument (polynomial set), w.r.t. a well-founded ordering. If we have equality, we move to comparing the second argument (set of pairs of polynomials):

- For the case $h_1 = h_2$, the lexicographic ordering clearly holds. The first argument does not change in the recursive call. The second argument decreases w.r.t the length of the tuples (which is well-founded).
- For the other case, however, it is not clear that the argument decreases. In fact we add elements to the first argument in the recursive call (and also to the second).

In the theory of Gröbner bases, it can be shown, for the **lc** (least common multiple) and **df** (substraction) discovered in the lazy thinking exploration, that if $h_1 \neq h_2$, then the leading power product of $h_1 - h_2$ is not a multiple of any of the leading power products already in the polynomial set.

Then, by Dickson's lemma, see [Dickson, 1913], there is no infinite such sequence of power products. Since to every polynomial added to the polynomial set there is a corresponding power product that verifies the condition in Dickson's lemma, this means that only finitely many new polynomials can be added to the polynomial set. This ensures termination (and is the canonical way to prove termination of Gröbner bases, see, for instance [Buchberger, 1970] - where Dickson's lemma was reinvented).

At this point the synthesis process is complete: We have synthesized by lazy thinking exploration the unknown algorithms in Algorithm["CPC scheme"], and in the synthesis process provided the proof of correctness, we also proved termination, which, in turn, ensures that the inference rules we used to prove properties of the algorithm scheme are correct.

4.6 Summary

Figure 4.3 outlines the synthesis process for our case study. Boxes represent exploration situations, ellipses represent explorations (actions). Full boxes and ellipses indicate situations and explorations that use or are used by lazy thinking exploration rounds in our case study, dashed boxes and ellipses do not involve lazy thinking, just exploration steps in the underlying theory, numbered circles indicate the steps in the case study, as follows:

1. The problem, "is finite Gröbner basis" was decomposed into 3 subproblems: "is Gröbner basis", "ideal equality" and "termination".
2. An inference rule to prove properties of the CPC algorithm scheme was added to the inference mechanism of each of the exploration situations. The correctness of this rule is conditioned by proving "termination".
3. In a preprocessing step, we proved a property of the CPC algorithm scheme. This property was added to the synthesis situation corresponding to "is Gröbner basis".
4. We then applied lazy thinking, which synthesized lc.
5. For "ideal equality", we applied lazy thinking (using the CPC inference rule), and obtained a (coupled) specification for GB and df;
6. Although lazy thinking did not produce df directly, this can be retrieved from the available knowledge base, following some exploration rounds in the underlying theory.
7. The subalgorithms obtained in the exploration rounds, from proving the correctness of "is Gröbner basis" and "ideal equality", together with Dickson's lemma are used to prove "termination".
8. With "termination" proved, the case study is complete. The application of the CPC inference rule at step 2 is correct.

The result of this synthesis case study is significant. We have managed to reinvent the notion of S-polynomials, the essential idea in the Gröbner bases theory, by lazy thinking.

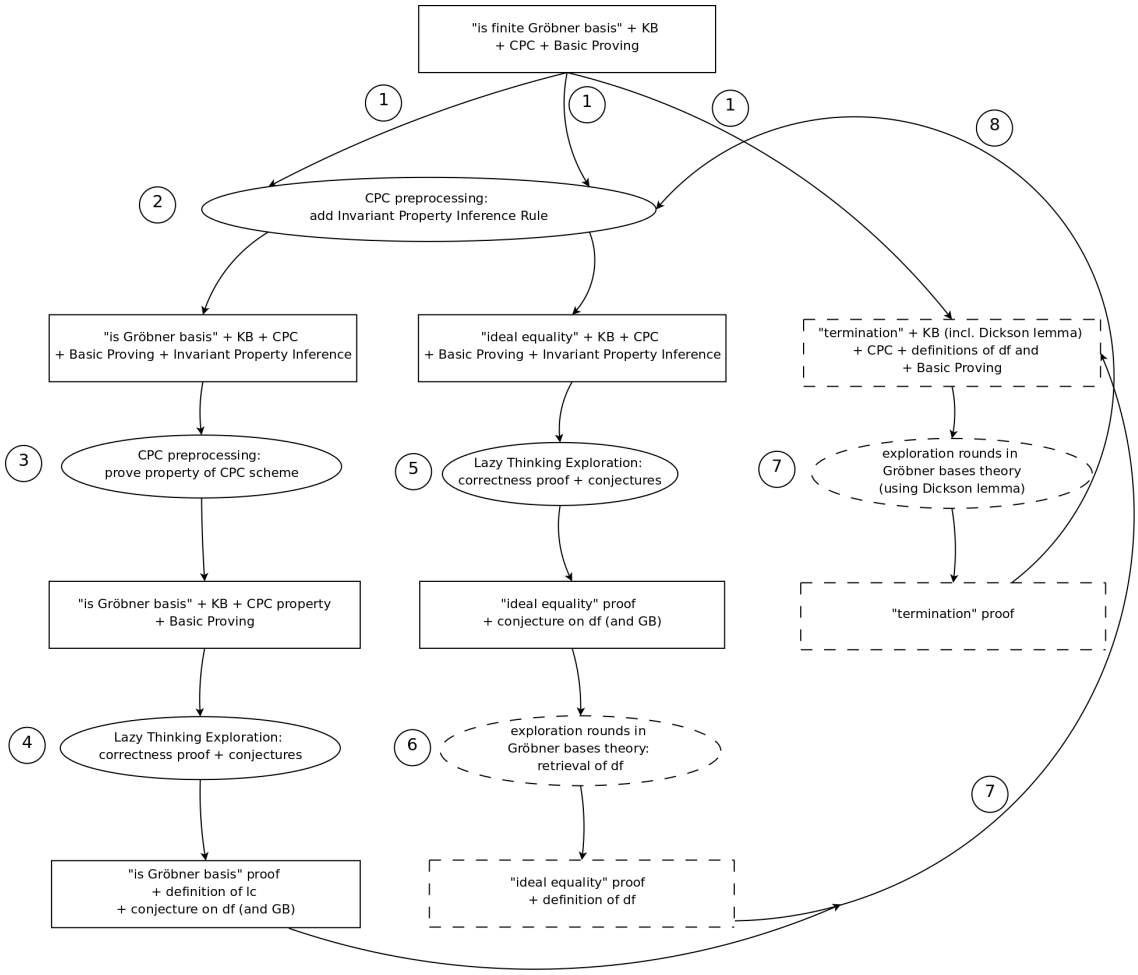

Figure 4.3: Gröbner bases synthesis by lazy thinking, workflow

CHAPTER 5

A Survey of Related Work

Lazy thinking, the method for algorithm synthesis which is the subject of this thesis, is part of an integral vision of systematic mathematical theory exploration, proposed by the first advisor, Bruno Buchberger, see [Buchberger, 2004a]. In the chapter dedicated to literature, we review and compare to lazy thinking, work in:

- *(computer-supported) discovery/invention* in mathematical theories, as the general context where lazy thinking plays its role,
- *algorithm synthesis*, as the essential activity carried out by lazy thinking, and
- computer supported exploration of *Gröbner bases theory*, as the most challenging domain, so far, where the method was applied.

5.1 Invention in Mathematical Theories

Historically, discovery/invention in mathematics assisted by computers was approached from an artificial intelligence point of view.

The AM program by Douglas Lenat, see [Lenat, 1976], was used to discover concepts and propose conjectures in elementary set and number theory. Concepts had a frame representation, each with 25 facets (such as the definition of a concept, algorithm for the concept, examples, relations to other concepts — specialization of, generalization of — conjectures). Initially, 115 elementary concepts (sets, bags), were provided. Tasks (such as specializations, generalizations, composition of functions, inverses, etc.) placed on an agenda of tasks were carried out according to one of 242 heuristics indicative of interestingness. AM was interactive, with the user guiding the invention according to

their interest. Some of the things discovered using AM were prime numbers, highly composite numbers and among the conjectures proposed were the fundamental theorem of arithmetic, Goldbach's conjecture.

In graph theory, several systems based on different (but complementary) ideas (enumeration, interaction, invariant manipulation, generation and selection, heuristic manipulation) were implemented to generate conjectures and new concepts. Notably, the Graffiti program [Fajtlowicz, 1988] produced conjectures of interest for graph theorists, which led to many published papers on the subject. A comparison of other programs for graph theory (GRAPE, Graph, GT, Ingrid, AutoGraphiX, Graffiti, etc.) can be found in the survey [Hansen and Mélot, 2002].

The HR program by Simon Colton, see [Colton, 2002], was used to discover concepts and conjectures in finite algebras, number and graph theory. Concepts are stored as data tables (examples) and corresponding axioms. HR production rules are then applied to obtain new concepts (specialize, generalize tables, combine tables) and conjectures (iff, implication, non-existence by checking data tables). To avoid a combinatorial explosion, several interestingness measures rank the concepts and conjectures. Conjectures are passed to a theorem prover (Otter), and if a proof is not successful, the generation of a counterexample is attempted (using MACE). HR's "greatest hits" include the addition of 14 sequences missing from the encyclopedia of integer sequences, the addition of 184 problems to the TPTP library.

The possible approaches to theorem discovery are discussed in [Sutcliffe et al., 2003]:

- the *inductive* approach, i.e. form a conjecture from a couple of examples, the approach mostly used by humans, unsound,
- the *generative* approach, an extension of the inductive approach, which is more efficient (as demonstrated by HR), but still unsound,
- the *manipulative* approach, i.e. modify known theorems by satisfiability preserving steps (which has the disadvantage of being artificial in nature, therefore uninteresting),
- the *deductive* approach, i.e. generate only logical consequences, and filter out only the interesting ones (which is considered a major challenge).

The MATHsAID system takes on the last approach: starting with some axiomatic description of a theory, logical consequences are generated, and then filtered out according to interestingness measures — nontriviality (done at the derivation stage), irredundancy (are all the hypotheses needed?) and simplicity (is the formula "simple"?), see [McCasland et al., 2006]. This approach yielded some encouraging results (i.e. theorems that typically appear in textbooks were identified interesting by the system) in number theory, set theory.

A different idea in the invention of mathematics is that of "fixing" nontheorems, in the spirit of Lakatos, see [Lakatos, 1976]: given a conjecture which turns out not to be a theorem several steps can be carried out to transform this into a theorem: modify the definition of concepts to exclude counterexamples (monster-barring), find properties that

lead to the counterexamples and eliminate them (piecemeal exclusion), etc. The TM system (using HR, MACE, Otter) was developed by Pease and Colton to use Lakatos style methods, see [Colton and Pease, 2004], [Pease, 2007] and investigations were carried out in number theory, group theory, etc.

Buchberger's scheme-based exploration model, of which the lazy thinking method is an important component (as discussed in Chapter 2, with examples) is different from the approaches mentioned above:

- it works in the frame of predicate logic (with deduction playing an essential role), close to the style of working mathematicians,
- it can be applied to any theory,
- schemes (concentrated mathematical knowledge) play an essential role in the efficiency of exploration (i.e. development, invention) of mathematical theories,
- the model provides methods for invention of concepts (definitions), propositions (conjectures), algorithms (via *lazy thinking*) by instantiation of schemes,
- and an important aspect is learning from failure (through failure analysis/conjecture generation).

Some similarities can be identified to the other approaches: heuristics similar to the production rules of HR can be used for the selection/instantiation of schemes, lazy thinking failure analysis and conjecture generation is similar in spirit to the Lakatos methods, although used with a different focus.

Initial case studies of theory exploration carried out in the scheme based model (natural numbers, well founded relations), see [Hodorog and Crăciun, 2007], [Crăciun and Hodorog, 2008], were encouraging: a good portion of the respective theories were reconstructed using schemes (e.g. the quotient remainder theorem, prime decomposition theorem — which included some steps similar to the Lakatos style analysis mentioned above, and where lazy thinking played an important role).

Prototypes of various components of the model (scheme instantiation, reasoners, lazy thinking) were implemented in *Theorema* by this author. Earlier work in *Theorema* included the implementation by others of a cascade mechanism for the invention of theorems similar to the cascade of lazy thinking, reported in [Buchberger et al., 2000].

However, the implementation of the scheme based model is in the early stages. A framework is needed for representation of theories (and their interaction) which to offer users (human or metaprovers) query and manipulation facilities. This will be built up on the existing functor constructs of *Theorema*, see [Buchberger et al., 1997], [Tomuta, 1998].

The implementation could benefit from some of the related work, e.g. using model checking to generate examples/counterexamples, allowing knowledge saturation (a la MATHsAID) in bottom-up explorations of certain notions.

5.2 Algorithm Synthesis

Algorithm synthesis is the process where, starting from a specification, an algorithm satisfying it is derived. Survey papers on algorithm synthesis, such as [Kreitz, 1998], [Basin et al., 2004] classify the existing approaches into:

- *Constructive/deductive synthesis (proofs as programs)*, where a (constructive) proof of the correctness statement (specification) is carried out, and the algorithm is extracted from this proof.
- *Transformational synthesis*, where the algorithm is obtained by the transformation of the specification into an (efficient) executable program,
- *Inductive synthesis*, where the challenge is to synthesize algorithms starting from incomplete specifications,
- *Knowledge based (scheme based) synthesis*, which makes use of the already well known algorithmic ideas to guide the synthesis process.

For the rest of this section, we give a short overview to the various approaches to algorithm synthesis, and we compare them to the lazy thinking method.

Algorithm synthesis is a very active area of research, and we restrict here to those which we believe are related and relevant to lazy thinking. Further pointers can be found by reading the survey articles which we indicate throughout this section.

5.2.1 Constructive/Deductive Synthesis

Synthesis from Tableaux

Manna and Waldinger, see [Manna and Waldinger, 1980],[Manna and Waldinger, 1992], start with a specification of the form $\forall a \exists z Q(a, z)$, and from the proof of this formula extract a term which stands for an algorithm satisfying this specification.

The proof is carried out in the frame of deductive tableaux. Each row in such a tableau may contain an assumption, a goal, and a corresponding term. A proof is a tableau with the last row containing $false$ in the assumption, or $true$ in the goal. Inference rules (splitting, resolution, equivalence replacement, skolemization, equality, induction) add new rows in the tableau, and in particular add new terms in the corresponding position. The term corresponding to the final goal is the desired algorithm.

When the synthesis is carried out in an inductive domain (with some sort of induction rule necessary to prove the specification theorem), different choices of induction may lead to different results. A choice of the well-founded relation used in the induction is delayed by introducing a generic relation. This is instantiated as the proof progresses with one of the well-founded relations already available in the theory.

The algorithms synthesized in this manner depend very much on the knowledge available (i.e. what formulae are known) and the proof (i.e. the choice and sequence of

inference steps), as these influence the terms being constructed (e.g. case analyses in the proof introduce conditionals or tests). This leads to very inefficient algorithms being synthesized.

Regarding the logic used in this approach, Manna and Waldinger do not require a strictly constructive logic. Rather, the only requirement is that the proof to be "sufficiently constructive" to allow the formation of the corresponding terms.

Synthesis by Proof Planning

Bundy, Kraan and Basin use proof planning [Bundy, 1988] was used to synthesize *pure logic programs* (i.e. in the body of the clauses — Horn bodies — only atoms, conjunction and disjunction of Horn bodies or existentially quantified Horn bodies are allowed), see [Kraan et al., 1996]. Restricting the language to this fragment of logic programs provides a unified frame to carry out the synthesis process.

Middle-out reasoning is used: A proof of the correctness statement is set up, with the Horn body of the desired program denoted by a metavariable. The proof is planned: since most of the programs are recursive in nature, an induction proof is set up: the base case and the induction step are set up, then *rippling* (a heuristic which tries to eliminate the differences between transforms induction step and the induction hypothesis so they can be used be used, e.g. by unification in a step called *fertilization*, see [Bundy et al., 1993]).

If the instantiations violate the pure logic programs, then they are considered specifications and *auxiliary synthesis* takes place, i.e. a subalgorithm is synthesized.

However, since in the case of synthesis the body of the algorithm is not known, it is not clear which induction should be applied (different induction schemes lead to different algorithms). Therefore this choice is postponed as much as possible: the parts that do not depend on the induction are planned first.

The constructor for the desired induction scheme is replaced by a metavariable which will be instantiated during the execution of the proof plan — this process is called *middle-out induction*. However, middle out induction introduces some problems (rippling may be non-terminating).

Middle out reasoning was used to synthesize algorithms on natural numbers (addition, multiplication), lists (insertion, max, append, reverse, etc.). However, as pointed out in [Kraan et al., 1996] some synthesis attempts, such as sorting, list partitioning, were not successful. The method was implemented in the *Clam* system. Lacey extended middle out synthesis to a higher order setting, and re-implemented it in λ-*Clam*, see [Lacey et al., 2000].

Armando, Smaill et al. used proof planning was also used to synthesize *functional programs*. The logic frame is that of Martin-Löf constructive type theory. Modulo the language frame, the approach is similar to that described above. Using this setting, several syntheses have been carried out: a decision procedure for propositional logic,

see [Armando et al., 1998], an unification algorithm, see [Armando et al., 1999].

The survey by Richardson, see [Richardson, 2002], provides pointers to other results in the use of proof planning in program synthesis.

Extraction of Algorithms from Proofs

Berger, Schwichenberg and others extract programs from intuitionistic proofs, by mapping every derivation in the proof to a term that "realizes" the formula being derived. Moreover, using the A-translation method, see [Berger and Schwichtenberg, 1995], classical proofs can be translated into intuitionistic proofs, and then algorithms can be extracted. One implementation where this can be carried out is the MinLog system. Nontrivial examples of algorithm extraction using this approach includes the Warshall algorithm, Dickson's lemma, see [Berger et al., 2001], or normalization algorithms, see [Berger et al., 2006].

Using a similar extraction mechanism implemented in Coq, see [Paulin-Mohring and Werner, 1993], Théry extracted a Gröbner bases algorithm from a formalization of the theory, see [Théry, 2001]. This will be further discussed in the next section.

Note, however, that in using this approach, although "programming" is for free – the algorithm is extracted from the proof – the effort goes into the formalization process. There is little invention in the program extraction process. This has to be explicitly formalized in the proof.

5.2.2 Synthesis by Transformation

This approach to synthesis is based on the program transformation ideas of Burstall and Darlington, see [Burstall and Darlington, 1977], where starting from a simple yet inefficient program, by the application of transformation rules, one arrives at a more efficient variant.

For program synthesis, the transformation starts from a specification of the type

$$\forall x \forall z (I[x] \Rightarrow (P(x,z) \Leftrightarrow O[x,z])),$$

where I is an input condition, P is the desired program, and O is the output condition.

The goal is to rewrite the output condition into equivalent or stronger formulae, possibly containing recursive calls of the algorithm:

$$\forall x \forall z (I[x] \Rightarrow (P(x,z) \Leftrightarrow O_f[x,z,P])),$$

where O_f is a formula that describes the body of the program. Used here is the notation of [Kreitz, 1998]. Program formation rules are then applied to transform the above

formula into an executable program. For the case of logic programs, program formation is almost straightforward.

The various approaches to transformational synthesis differ in the nature of the transformation rules, and the way these are applied.

Manna and Waldinger, see [Manna and Waldinger, 1979], use transformation rules of which some are domain dependent (e.g. properties of list for program on lists), others represent programming techniques, independent of domains (e.g. conditional introduction, recursion). The order of application of rules determines the final form of the program, and some strategies are provided for the application of rules (ordering rules, conditions for rules, backtracking).

In the approach of Dershowitz, see [Dershowitz, 1985], transformation rules are applied using a modified version of Knuth-Bendix completion procedure. Starting from the specifications, rules are extracted and then completed by Knuth-Bendix. Once complete, the rule system can easily be interpreted as a program.

Lau and Prestwitch, see [Lau and Prestwich, 1990], synthesize recursive logic programs by transforming *folding problems*. A specification is expressed as a first order logic equivalence formula (with some restrictions on the kind of formulae allowed). A folding problem is a triple containing the head of a program (same as the specification), the body, and a list of recursive calls to be used in the body. Initially, the body is described by what is essentially a metavariable, and this is gradually instantiated by the application of folding strategies: some for decomposition of a folding problem (DEFINITION, IMPLICATION), others for direct solving/instantiation (MATCH, MODUS PONENS).

In the LOPS system of Bibel, one of the first systems for algorithm synthesis, see [Bibel, 1980], transformation is done by the application of *strategies*: *GUESS-DOMAIN*, where a solution of the problem is guessed, which leads to the splitting of the program (the guess was either good or not), and *GET-REC*, which tries to introduce a recursion by rewriting towards an argument on which a recursive call can be made.

As pointed out in [Kreitz, 1998], [Basin et al., 2004], the deductive and transformational approaches to synthesis are closely related, and each can be cast into the other.

Another observation is that these approaches do not scale very well. Many of the synthesized examples are simple, and every time well known algorithmic principles are reinvented.

5.2.3 Inductive Synthesis

The inductive approach to program synthesis starts with incomplete input-output samples, and the goal is to generate (logic) programs whose clauses subsume these specifications.

One way to achieve this is to apply generalizations of the given examples, from simple syntactical generalizations to more complex ones, such as inverse resolution. However, sometimes these techniques can lead to overgeneralization (which can be dealt with by also including negative examples), and are not powerful enough to introduce recursion in the synthesized programs.

Flener and Deville, see [Flener and Deville, 1993], propose a hybrid approach, based on a synthesis strategy in steps, for *divide and conquer* programs: inductive synthesis steps such as syntactic or semantic generalization are combined with deductive synthesis steps (proofs-as-programs). The synthesis process is guided by the strategy, which essentially contains the *divide and conquer* idea (similar to algorithm schemes, discussed in the next section).

An approach more in the vein of transformational synthesis, by Hamfelt, Fischer Nilsson and Oldager, see [Hamfelt et al., 2001], applies composition rules to form programs (on lists) starting from incomplete specifications, in a top-down manner. Elementary composition rules form constants, identity and construct lists, operators are used to reorder arguments (*make*), introduce multiple clauses (*or*) or conjunctions in clauses (*and*), or introduce recursive calls (*foldl, foldr*).

The surveys [Flener and Yilmaz, 1999],[Basin et al., 2004] describe the various methods and results in inductive synthesis, as well as comparisons to the other approaches.

5.2.4 Scheme Based Synthesis

Some common shortcomings have been identified for the various approaches to synthesis discussed so far, such as the fact that they often re-invent well-known algorithmic ideas, or that they do not scale very well. Scheme based synthesis uses these ideas explicitly, to guide the synthesis process, thus dramatically decreasing the size of the search space.

We discuss below 2 major approaches to scheme based synthesis: for logic and for functional programs.

Scheme Based Logic Program Synthesis

Flener, Lau, Ornaghi, Richardson et. al., see [Flener et al., 1998], [Lau et al., 1999], [Flener et al., 2000], propose the use of schemes for the synthesis of pure logic programs.

A program schema represents the data and control flow of a class of programs, but does not contain their actual computations and data structures. The program schema is represented as a triple of *open programs* (templates), *specifications* of the predicates occurring in the template and *specification framework*, representing the background knowledge (theory).

Open programs are *steadfast* with respect to their specifications if they are correct when they are instantiated (or composed with other correct programs).

The steadfastness of each schema (such as divide and conquer) is proved *offline*. Program synthesis proceeds as follows: starting with a specification, select an algorithm scheme and instantiate it (choosing substitutions that lead to instantiations satisfying the specifications), until no open specifications are left. The transformations steps are proved to preserve correctness.

Synthesis strategies can guide the choice of open specifications, different choices leading to different programs. The big advantage of scheme based synthesis is that the difficult proof obligations are taken offline, the use of steadfast schemes ensures the correctness of the synthesized program.

Scheme Based Functional Program Synthesis

Smith (together with others) proposed the use of schemes for synthesis of functional programs. In fact, his work spans over more than two decades, and has produced some of the more important results in practical program synthesis.

Program are synthesized by transformations, in a framework based on a category of specifications, see [Pavlovic and Smith, 2003],[Smith, 1999].

Specifications are finite representations of some theory (including algorithmic aspects, and data structures). Morphisms between specifications are used to structure and parametrize the specifications, colimits are used to compose specifications and diagrams represent complex specifications. The framework contains a collection of techniques for the refinement of specifications and diagrams. Taxonomies of data structures and algorithmic ideas (schemes) are stored in this framework. Once the user classifies a specification, they can use these taxonomies to refine it.

Also in this , algorithm schemes as the structure of the program, together with formulae that ensure their correctness. The derivation of these conditions is done again offline, e.g. for divide-and-conquer in [Smith, 1985], or for global search algorithms in [Kreitz, 1996].

The framework is implemented in the systems Specware (for general purpose specification composing, code generation), Designware (which extends Specware with design taxonomies), Planware (generator of high-performance schedulers). An earlier implementation of the approach by Smith and collaborators was KIDS, see [Smith, 1990].

Bibel and others proposed the MAPS synthesis system, based on the Nuprl theorem prover, see [Bibel et al., 1998]. The approach is to combine several synthesis methods: Davis Putnam method for propositional logic, extraction from proofs, proof planning (inductive proving, rippling), strategy (scheme) based methods (similar to those of Smith mentioned above).

5.2.5 Comparing Lazy Thinking to Other Approaches to Algorithm Synthesis

Lazy thinking is a deductive scheme-based algorithm synthesis method that is similar in some ways with the approaches reviewed in this Section, but in the same time has some important original features:

- It is **similar** to deductive synthesis approaches, in that a proof of the correctness specification is set up and the algorithm is obtained from proving, but **different** in the method of obtaining the algorithm (specifications for subalgorithms vs. instantiation of metavariables), and, of course by the use of schemes. Although lazy thinking could be used without schemes, and we could obtain similar results (algorithms), we did not pursue this line of investigation (see Chapter 2, 2.2.6). Indeed the various results obtained by the deductive approaches are due to the presence of the right knowledge, i.e. implicit use of algorithmic (inductive structure) knowledge.
- *Failure analysis and conjecture generation* are the identifying features of lazy thinking, when compared to other approaches. Bundy and collaborators used failure, but this was for the generation of induction rules.
- The motivation for the use of schemes is **similar** to the other approaches.
- Preprocessing of schemes (like the example in Chapter 2, Section 2.3) ensures that in abstract settings (i.e. open, that cover many possible instantiations) the algorithm is correct, **similar** to steadfast schemes, or the schemes used in Smith's approach. However, lazy thinking is **different**, in that the correctness of schemes is proved *online*.
- As a result, the focus of lazy thinking is **different**. Both the scheme based approaches discussed focus on synthesis by transformation, where a framework and strategies for using algorithm schemes are available, and they were proved offline to ensure correctness of transformations. Lazy thinking is working on proving *online* correctness of programs (at various levels of abstraction). These results will be integrated in the future in a framework of theory exploration that will allow **similar** transformation steps.

5.3 Computer Supported Formal Gröbner Bases Theory

As far as we know, the work presented in this thesis is unique. The related work in formal Gröbner theory focuses on the formalization, in different systems, of Gröbner bases. We will now discuss the main approaches.

Théry formalized Gröbner bases theory, including Buchberger's algorithm in Coq, see [Théry, 2001], based on a presentation from [Geddes et al., 1992]. The structure of the theory is (not surprisingly) similar to that presented in our case study. Buchberger's algorithm was formulated, and its correctness proved by showing it returns a Gröbner

base and it preserves the ideal. Termination was reduced to Dickson's lemma. The algorithm was refined by considering Buchberger's criteria to avoid unnecessary reductions, reduced bases. An OCAML program was extracted from the Coq formalization.

Persson used part of Théry's formalization for his development of the Buchberger's algorithm in Coq, see [Persson, 2001]. However, most of the effort in this paper is dedicated to formalizing a proof of Dickson's lemma.

Another formalization was carried out in ACL2 by Medina-Bulo, Palomo-Lozano and others, see [Medina-Bulo et al., 2004]. The process of formalization is not much different from the previous approach (modulo system language particularities). However, in contrast to Coq, ACL2 allows the formulation of the executable algorithm directly in the language of the system (ACL2's programming language is COMMON LISP). The authors used previous formalizations of Dickson's lemma in ACL2 to prove termination.

Another relevant development is Schwarzweller's formalization of Gröbner bases theory in Mizar, see [Schwarzweller, 2005]. Mizar is a proof checker, so the user has to formalize the proofs that are then certified correct by the system. In contrast, both Coq and ACL2 provide some degree of automation (especially ACL2), or at least interactive proof development.

The significance of the formalizations in Coq and ACL2 is that they provide a verified implementation of a computer algebra algorithm. Also, for all the approaches the result of the formalization in a big body of machine processable mathematical text.

The focus of lazy thinking exploration is **different**: scheme-based algorithm synthesis. We showed that the method is capable re-invent the crucial concept of S-polynomials, starting from the Critical-Pair/Completion (CPC) algorithmic scheme (idea). In both the case of the Coq and ACL2 formalizations, all the ingredients (S-polynomials, algorithm) are explicitly provided as knowledge. Extracting the algorithm (for Coq) refers to the transformation into executable code from the logic of Coq.

Otherwise, the result of lazy thinking is **similar** to the others: we have an algorithm (which can be executed in the *Theorema* system) and its correctness proof. While the execution of this algorithm is likely to be extremely slow, current work in the *Theorema* group includes the development of a compiler for the language, which will significantly reduce the execution time.

CHAPTER 6

Conclusions

We summarize the work carried out in this thesis, then discuss future directions in which this work can be developed, and end with a discussion on the relevance of lazy thinking for its intended users.

6.1 Summary

We have presented in this thesis

- a particular formulation of the lazy thinking method in the context of a scheme-based exploration model,
- the implementation of lazy thinking in the *Theorema* system, and
- the application of this implementation to the synthesis of an algorithm for Gröbner bases,

starting from Bruno Buchberger's description of the method, and outline of the case study.

We described the main ingredients of the lazy thinking method (and their implementation):

- *the cascade* that organizes the exploration rounds consisting of successive attempts to prove that a specification (correctness theorem) holds for a solution proposed by the instantiation of an algorithm scheme,
- *the failure analyzer*, which extracts the failure situation (temporary knowledge and failing goal): the author's contribution was to specify a new analysis strategy

which allows extra temporary knowledge (certain universally quantified formulae),
- *the conjecture generation*, which uses the failure situation and generalization strategies to produce conjectures that allow the failure of the proof to be overcome: the author's contribution was to formulate a new generalization strategy, similar to *semantical pattern matching*.

We gave some simple examples of how lazy thinking can be used: with algorithm schemes (to synthesize specifications for the subalgorithms of the proposed scheme), or without (to synthesize constants). Also discussed was the situation when the specification cannot be satisfied.

One of the examples showed that lazy thinking can be used to *preprocess the scheme*: starting with a general form of an explicit problem, generate specifications for the subalgorithms of a scheme proposed as a solution. The results can then be used in every instantiation of the problem, and all instantiations of the algorithm scheme.

The case study, synthesis of a Gröbner bases algorithm, showcases many of the exploration techniques described in this thesis: introducing new inference rules, scheme preprocessing steps, various uses of our lazy thinking implementation (direct calls to the failure analyzer and conjecture generator and the influence of the failure analysis strategies in the generation of conjectures, in the situation when the user organizes manually the exploration cascade, automated exploration with the implemntation of the lazy thinking cascade).

We started with the formulation of the problem: given a polynomial set, we required (an algorithm to provide) the corresponding Gröbner basis. The background knowledge necessary to understand the problem was provided. We considered this to be developed in earlier exploration rounds in the scheme-based exploration model. Since the context allowed it (in the theory we have an appropriate reduction relation and corresponding operation) the Critical-Pair/Completion (CPC) algorithm scheme (instantiated for polynomials) was used to provide the structure of the algorithm to be synthesized.

We introduced an inference rule to prove properties of the CPC. This inference rule depends of the termination of the instantiation of the CPC scheme. We used the inference rule in exploration rounds that led to the proof of properties of CPC, then used these properties in the synthesis process. For one of the subproblems, the lazy thinking method leads directly to the definition of the function required. For the other, it is relatively easy to retrieve from the knowledge base functions that satisfy the generated requirements. The proof of termination does not involve the use of lazy thinking, and is standard for Gröbner bases theory. This ensures the correctness of the proofs produced using the CPC inference rule.

This completed the synthesis process: This case study showed that, using the method of lazy thinking, we can re-invent the essential idea of *S-polynomials*.

6.2 Future Work

We see several directions in which the work presented in this thesis can be extended.

We did not yet consider efficiency specifications. These could be included in the formulation of the problems, as suggested in the early synthesis work by Manna and Waldinger. Also, the efficiency of algorithmic ideas could be investigated, by applying lazy thinking to prove efficiency properties of abstract schemes, in preprocessing steps.

Future work could benefit from some of the related approaches in the field of synthesis. Smith provided a taxonomy of algorithmic ideas, these are natural case studies for our method. The investigation of algorithmic ideas which were not considered so far will contribute to the development of algorithm scheme libraries for *Theorema*, which in turn will benefit theory exploration case studies, and in turn the development of mathematical knowledge in *Theorema*.

Techniques used in inductive synthesis may suggest new generalization strategies for our conjecture generator.

The addition of model theoretic techniques, such as those used by Colton for invention in mathematical theories could benefit lazy thinking explorations, by providing extra information in the form of examples or counterexamples.

Lazy thinking is one of the exploration techniques that form Buchberger's scheme based exploration model. This was proposed recently, and is being investigated by carrying out case studies in various theories (natural numbers, tuples, Gröbner bases, etc.). A framework for carrying out and managing the exploration is needed, and these case studies help identify the issues and design requirements for such a framework: how to represent theories, and their relations (the functors in *Theorema* seem to be a suitable mechanism), queries, mathematical knowledge retrieval.

Related to our main case study, future work can include the exploration by lazy thinking of the theories where the CPC idea can be applied (resolution, word problems in universal algebras), improvements of the algorithms for Gröbner bases (synthesizing criteria to avoid unnecessary reductions, reduced Gröbner bases).

6.3 The Relevance of Lazy Thinking

Lazy thinking is a general method for problem solving in predicate logic, so it is not tied to its implementation in *Theorema*.

Lazy thinking and the more general context of scheme-based theory exploration provide:

- To the "working mathematician": A method to identify problems and look for their solutions.
- To the "aspiring mathematician" (student): A way to understand algorithms, how and why they work (and a tool to decide whether they work), in the larger context

of understanding how mathematics works.

These remarks are based on this author's own experience while working on the implementation and use of the method in case studies and we believe that this thesis provides evidence of that. We think the method is well worth adding to the array of tools used by working (computer) mathematicians. Carrying out further case studies may provide insight into the process of invention, allow lazy thinking to tackle successfully new problems and provide new solutions.

Bibliography

[Armando et al., 1998] Armando, A., Gallagher, J., Smaill, A., and Bundy, A. (1998). Automating the Synthesis of Decision Procedures in a Constructive Metatheory. *Annals of Mathematics and Artificial Intelligence*, 22(3-4):259–279.

[Armando et al., 1999] Armando, A., Smaill, A., and Green, I. (1999). Automatic Synthesis of Recursive Programs: The Proof-Planning Paradigm. *Automated Software Engineering: An International Journal*, 6(4):329–356.

[Basin et al., 2004] Basin, D., Deville, Y., Flenner, P., Hamfelt, A., and Nilsson, J. (2004). *Program Development in Computational Logic*, volume 3049 of *LNCS*, chapter Synthesis of Programs in Computational Logic, pages 30–65. Springer Verlag.

[Benzmller et al., 1997] Benzmller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Melis, E., Meier, A., Schaarschmidt, W., Siekmann, J., and Sorge, V. (1997). Omega: Towards a Mathematical Assistant. In *Proceedings of 14th International Conference on Automated Deduction (CADE-14)*, pages 252–255. Springer.

[Berger et al., 2006] Berger, U., Berghofer, S., Letouzey, P., and Schwichtenberg, H. (2006). Program Extraction from Normalization Proofs. *Studia Logica*, 82:25–49.

[Berger and Schwichtenberg, 1995] Berger, U. and Schwichtenberg, H. (1995). Program Extraction from Classical Proofs. In *Logic and Computational Complexity, International Workshop LCC'94, Indianapolis, IN, USA, October 1994*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag.

[Berger et al., 2001] Berger, U., Schwichtenberg, H., and Seisenberger, M. (2001). The Warshall Algorithm and Dickson's Lemma: Two Examples of Realistic Program Extraction. *J. Autom. Reason.*, 26(2):205–221.

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer.

[Bibel, 1980] Bibel, W. (1980). Syntax-Directed, Semantics-Supported Program Synthesis. *Artif. Intell.*, 14(3):243–261.

[Bibel et al., 1998] Bibel, W., Korn, D. S., Kreitz, C., Kurucz, F., Otten, J., Schmitt, S., and Stolpmann, G. (1998). A multi-level approach to program synthesis. In Fuchs, N. E., editor, *LOPSTR: Logic Programming Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 10-12, 1997, Proceedings*, volume 1463 of *Lecture Notes in Computer Science*, pages 1–27. Springer.

[Buchberger, 1965] Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria. (English translation Journal of Symbolic Computation 41 (2006) 475–511).

[Buchberger, 1970] Buchberger, B. (1970). Ein algorithmisches Kriterium fuer die Loesbarkeit eines algebraischen Gleichungssystems (An Algorithmic Criterion for the Solvability of Algebraic Systems of Equations). *Aequationes mathematicae*, 3:374–383. (english transl.: B. Buchberger, F. Winkler: Groebner Bases and Applications, Proc. of the International Conference "33 Years of Groebner Bases", 1998, RISC, Austria, London Math. Society Lecture Note Series 251, Cambridge Univ. Press, 1998, pp.535 -545).

[Buchberger, 1985] Buchberger, B. (1985). *Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory.* Reidel Publishing Company, Dodrecht - Boston - Lancaster.

[Buchberger, 1987] Buchberger, B. (1987). History and Basic Features of the Critical-Pair/Completion Procedure. *Journal of Symbolic Computation*, 3(1/2):3–38. (Earlier version appeared in: Proceedings of the Conference on Rewrite Technique and Applications, Dijon, May 1985, Lecture Notes in Computer Science, Vol. 202, Springer, 1985, pp. 1-45).

[Buchberger, 1991] Buchberger, B. (1991). Logic for Computer Science. RISC, Johannes Kepler University, Austria, Lecture notes.

[Buchberger, 1998] Buchberger, B. (1998). Introduction to Gröbner Bases. In Buchberger, B. and Winkler, F., editors, *Gröbner Bases and Applications*, volume 251 of *London Mathematical Society Lectures Notes Series*, pages 3–31. Cambridge University Press, Johannes Kepler University of Linz.

[Buchberger, 2001] Buchberger, B. (2001). The PCS Prover in Theorema. In Moreno-Diaz, R., Buchberger, B., and Freire, J., 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 –. Las Palmas de Gran Canaria, Copyright: Springer - Verlag Berlin.

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

[Buchberger, 2004a] Buchberger, B. (2004a). Algorithm Supported Mathematical Theory Exploration: A Personal View and Stragegy. In Buchberger, B. and Campbell, J., editors, *Proceedings of AISC 2004 (7 th International Conference on Artificial Intelligence and Symbolic Computation)*, volume 3249 of *Springer Lecture Notes in Artificial Intelligence*, pages 236–250. RISC, Johannes Kepler University, Austria, Springer, Berlin-Heidelberg.

[Buchberger, 2004b] Buchberger, B. (2004b). Towards the Automated Synthesis of a Groebner Bases Algorithm. *RACSAM - Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science)RACSAM - Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science), Serie A: Mathematicas*, 98(1):65–75.

[Buchberger and Caprotti, 2001] Buchberger, B. and Caprotti, O. (2001). First International Workshop on Mathematical Knowledge Management (MKM 2001). Technical report, RISC, Johannes Kepler University, Austria.

[Buchberger and Crăciun, 2004a] Buchberger, B. and Crăciun, A. (2004a). Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In Kamareddine, F., editor, *Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003.*, volume 93 of *Electronic Notes in Theoretical Computer Science*, pages 24–59.

[Buchberger and Crăciun, 2004b] Buchberger, B. and Crăciun, A. (2004b). Algorithm Synthesis by Lazy Thinking: Using Problem Schemes. In Petcu, D., Negru, V., Zaharie, D., and Jebelean, T., editors, *Proceedings of SYNASC 2004, 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 90–106, Timisoara, Romania.

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

[Buchberger et al., 2000] Buchberger, B., Dupre, C., Jebelean, T., Kriftner, F., Nakagawa, K., Vasaru, D., and Windsteiger, W. (2000). The Theorema Project: A Progress Report. In Kerber, M. and Kohlhase, M., 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.

[Buchberger et al., 1997] Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuta, E., and Vasaru, D. (1997). A Survey of the Theorema project. In Kuechlin, W.,

editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997)*, pages 384–391. ACM Press.

[Buchberger and Nakagawa, 2004] Buchberger, B. and Nakagawa, K. (2004). Mathematical Knowledge Editor: A Research Plan. Technical report, Johannes Kepler University Linz. Spezialforschungsbereich SF013 "Scientific Computing", FWF (Austrian National Science Foundation).

[Buchberger and Winkler, 1998] Buchberger, B. and Winkler, F. (1998). *Gröbner Bases and Applications*, volume 251 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press.

[Buchberger and Zapletal, 2006] Buchberger, B. and Zapletal, A. (2006). Gröbner Bases Bibliography. `http://www.ricam.oeaw.ac.at/Groebner-Bases-Bibliography/`.

[Bundy, 1988] Bundy, A. (1988). The Use of Explicit Plans to Guide Proofs. In Lusk, E. and Overbeek, R., editors, *Proceedings of CADE-9*, number 310 in Lecture Notes in Computer Science, pages 111–120. Springer-Verlag.

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence*, 62(2):185–253.

[Bundy et al., 1990] Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E., editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag.

[Burstall and Darlington, 1977] Burstall, R. M. and Darlington, J. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67.

[Buss, 1998] Buss, S. R. (1998). *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, chapter An Introduction to Proof Theory, pages 1–78. Elsevier Sicence B.V.

[Carlisle, 2000] Carlisle, D. (2000). OpenMath, MathML and XSL. *SIGSAM Bulletin*, 34(2):6–11.

[Church, 1936] Church, A. (1936). A Note on the "Entscheidungsproblem". *The Journal of Symbolic Logic*, 1(I):40–41.

[Colton, 2002] Colton, S. (2002). *Automated Theory Formation in Pure Mathematics*. Springer-Verlag.

[Colton and Pease, 2004] Colton, S. and Pease, A. (2004). Lakatos-Style Automated Theorem Modification. In López de Mántaras, R. and Saitta, L., editors, *ECAI: Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 977–978. IOS Press.

[Constable et al., 1986] Constable, R. L., Allen, S. F., Cleaveland, W., Cremer, J., Harper, R., Howe, D. J., Knoblock, T. B., Mendler, N., Panangaden, P., Sasaki, J. T., and Smith, S. F. (1986). *Implementing Mathematics with the {N}uprl Development System*. Prentice-Hall, NJ.

[Cox et al., 2005] Cox, D., Little, J., and O'Shea, D. (2005). *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer.

[Crăciun and Hodorog, 2008] Crăciun, A. and Hodorog, M. (2008). Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In Petcu, D., Negru, V., Zaharie, D., and Jebelean, T., editors, *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC07)*. IEEE Computer Society. to appear as IEEE volume.

[Curry, 1942] Curry, H. B. (1942). The Combinatory Foundations of Mathematical Logic. *Journal of Symbolic Logic*, 7(2):49–64.

[Davis, 2001] Davis, M. (2001). The Early History of Automated Deduction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 1, pages 3–15. Elsevier Amsterdam London New York Oxford Paris Shannon Tokyo.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215.

[Dershowitz, 1985] Dershowitz, N. (1985). Synthesis by Completion. In Joshi, A., editor, *IJCAI Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, August 1985.*, pages 208–214. Morgan Kaufmann.

[Dickson, 1913] Dickson, L. E. (1913). Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors. *Am. J. Math.*, 35:413–422.

[Duff and Watson, 1998] Duff, I. and Watson, G., editors (1998). *The State of the Art in Numerical Analysis*, volume 63 of *The Institute of Mathematics and its Applications, New Series*. Oxford University Press.

[Fajtlowicz, 1988] Fajtlowicz, S. (1988). On Conjectures of Graffiti. *Discrete Mathematics*, 72(23):113–118.

[Farmer et al., 1996] Farmer, W., Guttman, J., and Thayer Fábrega, F. (1996). IMPS: an Updated System Description. In McRobbie, M. and Slaney, J., editors, *Automated Deduction–CADE-13*, volume 1104, pages 298–302. LNCS.

[Flener and Deville, 1993] Flener, P. and Deville, Y. (1993). Logic Program Synthesis from Incomplete Specifications. *J. Symbolic Computation*, (15):775–805.

[Flener et al., 1998] Flener, P., Lau, K.-K., and Ornaghi, M. (1998). On Correct Program Schemas. *Lecture Notes in Computer Science*, 1463:128–147.

[Flener et al., 2000] Flener, P., Lau, K.-K., Ornaghi, M., and Richardson, J. (2000). An Abstract Formalization of Correct Schemas for Program Synthesis. *J. Symb. Comput.*, 30(1):93–127.

[Flener and Yilmaz, 1999] Flener, P. and Yilmaz, S. (1999). Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects. *Journal of Logic Programming*, 41(2-3):141–195.

[Frege, 1879] Frege, G. (1879). *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle.

[Frege, 1893] Frege, G. (1893). *Grundgesetze der Arithmetik*, volume I. Verlag Herman Pohle.

[Frege, 1903] Frege, G. (1903). *Grundgesetze der Arithmetik*, volume II. Verlag Herman Pole.

[Geddes et al., 1992] Geddes, K., Czapor, S., and Labahn, G. (1992). *Algorithms for Computer Algebra*. Kluwer Academic Publishers.

[Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198.

[Gordon and Melham, 1993] Gordon, M. and Melham, T., editors (1993). *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

[Grabmeier et al., 2003] Grabmeier, J., Kaltofen, E., and Weispfenning, V., editors (2003). *Computer Algebra Handbook*. Springer-Verlag New York, Inc., New York, NY, USA.

[Hamfelt et al., 2001] Hamfelt, A., Fischer Nilsson, J., and Oldager, N. (2001). Logic Program Synthesis as Problem Reduction Using Combining Forms. *Automated Software Engineering*, 8(2):421–448.

[Hansen and Mélot, 2002] Hansen, P. and Mélot, H. (2002). Computers and Discovery in Algebraic Graph Theory. *Linear Algebra and its Applications*, (356):211 – 230.

[Hilbert, 1900] Hilbert, D. (1900). Mathematische Probleme. *Nachrichten von der Königlichten Gesellschaft der Wissenschaften zu Göttingen, Math.-Phys. Klasse*, pages 253–297.

[Hodorog and Crăciun, 2007] Hodorog, M. and Crăciun, A. (2007). Scheme-Based Systematic Exploration of Natural Numbers. In *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, September 26-29, 2006*, pages 23–34. IEEE Computer Society.

[Kaufmann et al., 2000] Kaufmann, M., Manolios, P., and Moore, J. S. (2000). *Computer–Aided Reasoning: An Approach*. Kluwer Academic Publishers.

[Knuth, 1984] Knuth, D. E. (1984). *The TeXbook*. Addison–Wesley, Reading, Massachusetts.

[Knuth and Bendix, 1970] Knuth, D. E. and Bendix, P. B. (1970). Simple Word Problems in Universal Algebra. In J. Leech, editor, *Proc. of the Conf. of Computational Problems in Abstract Algebra, Oxford, 1967*. Pergamon Press.

[Kraan et al., 1996] Kraan, I., Basin, D. A., and Bundy, A. (1996). Middle-Out Reasoning for Synthesis and Induction. *J. Autom. Reasoning*, 16(1-2):113–145.

[Kreitz, 1996] Kreitz, C. (1996). Formal Mathematics for Verifiably Correct Program Synthesis. *Journal of the IGPL*, 4(1):75–94.

[Kreitz, 1998] Kreitz, C. (1998). *Automated Deduction - A Basis for Applications*, chapter Program Synthesis, pages 105–134. Kluwer Academic Publishers.

[Kutsia and Buchberger, 2004] Kutsia, T. and Buchberger, B. (2004). Predicate Logic with Sequence Variables and Sequence Function Symbols. In Asperti, A., Bancerek, G., and Trybulec, A., editors, *Proceedings of the 3rd International Conference on Mathematical Knowledge Management, MKM'04*, volume 3119 of *Lecture Notes in Computer Science*, pages 205–219, Bialowieza, Poland. Springer Verlag.

[Lacey et al., 2000] Lacey, D., Richardson, J., and Smaill, A. (2000). Logic Program Synthesis in a Higher-Order Setting. In et al, J. L., editor, *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 87–100. Springer Verlag.

[Lakatos, 1976] Lakatos, I. (1976). *Proofs and Refutations*. Cambridge University Press.

[Lamport, 1986] Lamport, L. (1986). *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts.

[Lau et al., 1999] Lau, K.-K., Ornaghi, M., and Tarnlund, S.-A. (1999). Steadfast Logic Programs. *Journal of Logic Programming*, 38(3):259–294.

[Lau and Prestwich, 1990] Lau, K.-K. and Prestwich, S. D. (1990). Top-down Synthesis of Recursive Logic Procedures from First-order Logic Specifications. In Warren, D. H. D. and Szeredi, P., editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 667–684, Jerusalem. The MIT Press.

[Lenat, 1976] Lenat, D. (1976). *AM: An Artificial Intelligence Approach to Discovery in Mathematics*. PhD thesis, Stanford University.

[Lenzen, 2004] Lenzen, W. (2004). Leibniz's Logic. In Gabbay, D. and Woods, J., editors, *The Rise of Modern Logic - From Leibniz to Frege (Handbook of the History of Logic - Vol. 3)*, pages 1–83. Elsevier, Amsterdam.

[Manna and Waldinger, 1979] Manna, Z. and Waldinger, R. (1979). Synthesis: Dreams - Programs. *IEEE Trans. Software Eng.*, 5(4):294–328.

[Manna and Waldinger, 1980] Manna, Z. and Waldinger, R. (1980). A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121.

[Manna and Waldinger, 1992] Manna, Z. and Waldinger, R. (1992). Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704.

[McCarthy, 1958] McCarthy, J. (1958). Programs with Common Sense. In *Symposium on Mechanization of Thought Processes*, Teddington, England. National Physical Laboratory.

[McCasland et al., 2006] McCasland, R. L., Bundy, A., and Smith, P. F. (2006). Ascertaining Mathematical Theorems. *Electr. Notes Theor. Comput. Sci.*, 151(1):21–38.

[Medina-Bulo et al., 2004] Medina-Bulo, I., Palomo-Lozano, F., Alonso-Jiménez, J. A., and Ruiz-Reina, J.-L. (2004). Verified Computer Algebra in ACL2. gröbner Bases Computation. In Buchberger, B. and Campbell, J. A., editors, *AISC: Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings*, pages 171–184.

[Nakagawa et al., 2004] Nakagawa, K., Nomuro, A., and Suzuki, M. (2004). Extraction of Logical Structure from Articles in Mathematics. In *Proceedings of the Third International Conference on Mathematical Knowledge Management (MKM 2004)*, number 3119 in LNCS, Bialowieza, Poland, September 19-21. Springer Heidelberg-Berlin-New York.

[Nederpelt et al., 1994] Nederpelt, J., Geuvers, J., and de Vrijer, R., editors (1994). *Selected Papers on Automath*, volume 133 of *Studies in Logic*. North-Holland.

[Newell et al., 1995] Newell, A., Shaw, J., and Simon, H. (1995). Empirical Explorations with the Logic Theory Machine: a Case Study in Heuristics. In *Computers & thought*, pages 109–133. MIT Press, Cambridge, MA, USA.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer.

[Paulin-Mohring and Werner, 1993] Paulin-Mohring, C. and Werner, B. (1993). Synthesis of ML Programs in the System Coq. *J. Symb. Comput.*, 15(5-6):607–640.

[Pavlovic and Smith, 2003] Pavlovic, D. and Smith, D. R. (2003). Software Development by Refinement. In Aichernig, B. K. and Maibaum, T. S. E., editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support*, volume 2757, pages 267–286. Springer.

[Pease, 2007] Pease, A. (2007). *A Computational Model of Lakatos-style Reasoning*. PhD thesis, School of Informatics, University of Edinburgh.

[Persson, 2001] Persson, H. (2001). An Integrated Development of Buchberger's Algorithm in Coq. Technical Report 4271, INRIA Sophia Antipolis.

[Pfenning, 1996] Pfenning, F. (1996). The Practice of Logical Frameworks. In Kirchner, H., editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden. Springer-Verlag LNCS 1059.

[Piroi, 2004] Piroi, F. (2004). *Tools for Using Automated Provers in Mathematical Theory Exploration*. PhD thesis, Research Institute for Symbolic Computation, Johannes Kepler University of Linz, Austria. Also available as RISC Tech Report no. 04-12.

[Richardson, 2002] Richardson, J. (2002). Proof Planning and Program Synthesis: a Survey. In *Logic-Based Program Synthesis: State of the Art and Future Trends, Papers from 2002 AAAI Spring Symposium*, number SS-02-05, pages 7–12. AAAI Press.

[Robinson, 1965] Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41.

[Rudnicki, 1992] Rudnicki, P. (1992). An Overview of the Mizar Project. In *Proceedings of the Workshop on Types for Proofs and Programs*, Bastad, Sweden. Chalmers University for Technology.

[Schönfinkel, 1967] Schönfinkel, M. (1967). On the Building Blocks of Mathematical Logic. In van Heijenoort, J., editor, *From Frege to Gödel: a source book in mathematical logic, 1879-1931*. Harvard University Press. original title, "er die Bausteine der mathematischen Logik," 1924.

[Schwarzweller, 2005] Schwarzweller, C. (2005). Groebner Bases - Theory Refinement in the Mizar System. In *4th International Conference on Mathematical Knowledge Management, Proceedings*, number 3863 in Lecture Notes in Artificial Intelligence, pages 299–314. Springer Verlag.

[Smith, 1985] Smith, D. R. (1985). Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, (27):43–96.

[Smith, 1990] Smith, D. R. (1990). KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043.

[Smith, 1999] Smith, D. R. (1999). Mechanizing the Development of Software. In Broy, M., editor, *Calculational System Design, Proceedings of the International Summer School Marktoberdorf*, NATO ASI Series, Amsterdam. IOS Press.

[Sutcliffe et al., 2003] Sutcliffe, G., Gao, Y., and Colton, S. (2003). A Grand Challenge of Theorem Discovery. In the proceedings of the CADE workshop on Challenges and Novel Applications for Automated Reasoning.

[Tanenbaum, 2006] Tanenbaum, A. S. (2006). *Structured Computer Organization*. Prentice Hall.

[Théry, 2001] Théry, L. (2001). A Machine-Checked Implementation of Buchberger's Algorithm. *J. Autom. Reason.*, 26(2):107–137.

[Tomuta, 1998] Tomuta, E. (1998). *An Architecture for Combining Provers and its Applications in the Theorema System*. PhD thesis, RISC, Johannes Kepler University of Linz.

[Turing, 1936] Turing, A. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265. http://www.turingarchive.org/browse.php/B/12.

[Whitehead and Russel, 1910] Whitehead, A. and Russel, B. (1910). *Principia Mathematica*, volume 1. Cambridge University Press, Cambridge.

[Whitehead and Russel, 1912] Whitehead, A. N. and Russel, B. (1912). *Principia Mathematica*, volume 2. Cambridge University Press, Cambridge.

[Whitehead and Russel, 1913] Whitehead, A. N. and Russel, B. (1913). *Principia Mathematica*, volume 3. Cambridge University Press, Cambridge.

[Winkler, 1996] Winkler, F. (1996). *Polynomial Algorithms in Computer Algebra*. Springer.

[Wolfram, 2003] Wolfram, S. (2003). *The Mathematica Book*. Wolfram Media, Incorporated.

Curriculum Vitae

Adrian Crăciun

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Email: acraciun@risc.uni-linz.ac.at

PERSONAL DATA

Date and place of birth: 22.02.1977, Ineu, Romania.
Citizenship: Romanian.
Personal status: Married.

EDUCATION

2000-, PhD study at Research Institute for Symbolic Computation, Johannes Kepler University, Linz, supervised by Prof. Dr. Bruno Buchberger.
• PhD thesis: *Algorithm Synthesis in Gröbner Bases Theory*, 2008.
1999-2000, Advanced Studies, Computer Science Department, Faculty of Mathematics, West University, Timişoara.
• Dissertation: *Constraint Logic Programming* (in Romanian).
1995-1999, Computer Science studies, Computer Science Department, Faculty of Mathematics, West University, Timişoara
• Diploma Thesis: *Multimedia Databases* (in Romanian).
1991-1995, "Mihai Viteazul" Highschool, Ineu, jud. Arad, Romania.
• Graduation Exam (Bacalaureat): June 1995.

SCIENTIFIC EXPERIENCE

Scientific Interests

- Automated theorem proving.
- Mathematical knowledge management.
- Program synthesis and verification.
- Logic programming, functional programming.
- Computer algebra.
- Logic, set theory.

Working Groups

Theorema: Member of the *Theorema* group at RISC-Linz from March 2001.

SysteMaThEx: Principal researcher in the *SysteMaThEx* research group at e-Austria Research Institute, Timisoara, from October 2004 to August 2007.

Phasetrans: Researcher, phase transitions in computational complexity, with Dr. Gabriel Istrate.

PROJECT EXPERIENCE

Calculemus

- Sept. 2001 - Sept. 2003, Young Visiting Researcher RISC-Linz.
- Sept. 2003 - Dec. 2003, Young Visiting Researcher Centre for Intelligent Systems and their Applications (CISA), University of Edinburgh.
- Jan. 2004 - Aug. 2004, Young Visiting Researcher RISC-Linz.

FP6 MERG-CT-2004-012718: SysteMaThEx. Scientific Coordinator for the Marie Curie Reintegration Grant Project MERG-CT-2004-012718 SysteMaThEx: "Systematic Mathematical Theory Exploration with the Theorema System: Case Studies", (Institute e-Austria, Timişoara, Romania).

SEPROPI. Team Coordinator (Theory Exploration) for the Romanian National University Research Council Project SEPROPI: "Studies and systematic exploration of parallel iterative processes", (Institute e-Austria, May 2006 -December 2007, project leader Prof. Dr. Ştefan Măruşter).

Phasetrans. Researcher (e-Austria Research Institute, from September 2007, project leader Dr. Gabriel Istrate).

PUBLICATIONS

Articles in Journals

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

Refereed Articles in Conference Proceedings

□ A. Crăciun and B. Buchberger, *Proving the Correctness of the Merge-Sort Algorithm with Theorema*, In: Symbolic and Numeric Algorithms for Scientific Computing, Proceedings of the SYNASC02 Workshop, Oct. 9-12, 2002, Timişoara, Romania, Editura Mirton, Timişoara, 2002.
□ B. Buchberger and A. Crăciun, *Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema*, Proceedings of the Mathematical Knowledge Management Symposium, Electronic Notes in Theoretical Computer Science, Volume 93, 18 February 2004, pp 24-59.
□ B. Buchberger and A. Crăciun, *Algorithm Synthesis by Lazy Thinking: Using Problem Schemes*, Proceedings of the 6th SYNASC, Sept. 26-30, 2004, Timişoara, Romania, Mirton Publishing, Timişoara, 2004.
□ M. Hodorog and A. Crăciun, *Scheme-Based Systematic Exploration of Natural Numbers*, Proceedings of the 8th SYNASC, Sept. 26-19, 2006, Timişoara, Romania, IEEE Computer Society, 2334, 2007.
□ A. Crăciun and M. Hodorog.: *Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration*, Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC07), IEEE Computer Society, West University of Timisoara, Romania, Eds: Petcu, D., Negru, V., Zaharie, D., and Jebelean, T., 2008

Articles in Conference Proceedings

□ A. Crăciun, *The Sequence Provers in Theorema*, Calculemus 2002, Work in Progress Papers, Seki-Report Nr. SR-02-04, Universität des Saarlandes.
□ A. Crăciun and B. Buchberger, *Functional Program Verification with Theorema*, Computer Aided Verification of Information Systems: A Practical Industry-Oriented Approach - CAVIS03, Workshop, e-Austria Institute, Timişoara, February 12, 2003.

Technical Reports

□ B. Buchberger, T. Jebelean, W. Windsteiger, T. Kutsia, K. Nakagawa, J. Robu, F. Piroi, A. Crăciun, N. Popov, G. Kusper, M. Rosenkranz, L. Kovacs, C. Kocsis . *F 1302: THEOREMA: Proving, Solving, and Computing in the Theory of Hilbert Spaces.* In: Special Research Program (SFB) F 013, Numerical and Symbolic Scientific Computing, Proposal for Continuation, Part II: Proposal, P. Paule, U. Langer (ed.), pp. 58-73. October 2003. Johannes Kepler University Linz, Austria.
□ A. Crăciun, B. Buchberger. *Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking.* Technical report no. 04-16 in RISC Report Series, University of Linz, Austria. October 2004.
□ A. Crăciun, B. Buchberger. *Preprocessed Lazy Thinking: Synthesis of Sorting Algorithms.* Technical report no. 04-17 in RISC Report Series, University of Linz, Austria. October 2004.
□ A. Crăciun. *An Introduction to the SysteMaThEx Project*, IeAT Technical Report 05-03, Institute e-Austria Timişoara, Romania, 2005.

Contributed Conference/Miscellaneous Talks

□ A. Crăciun and B. Buchberger, *The Lazy Thinking Paradigm: Top-Down Theory Exploration using Theorema*, poster at the Calculemus Midterm Review Meeting, University of Saarlandes, Saarbruecken, Germany, March 30-April 1, 2003.
□ A. Crăciun and B. Buchberger, *Mathematical Knowledge Management for Reasoning about Programs: Tuples and Sorting*, SFBStatus Seminar, Strobl, Austria, April 24-26 2003.
□ A. Crăciun and B. Buchberger, *Theory Exploration by Lazy Thinking: A Case Study*, Omega- *Theorema* Workshop, Hagenberg, Austria, May 25-27, 2003.
□ A. Crăciun, *An Overview of THEOREMA / Program Synthesis with THEOREMA Support*, CISA-Symposium Talk, Edinburgh, September 25, 2003.
□ A. Crăciun, *THEOREMA: An Overview*, Scottish Theorem Provers Workshop, University of Strathclyde, Glasgow, 19 Dec 2003.
□ A. Crăciun, *An Implementation of Groebner Synthesis in Theorema,* Special Semester on Groebner Bases and Related Methods, Workshop C: Formal Groebner Bases Theory, Johannes Kepler University/Radon Institute for Computational and Applied Mathematics, Linz March 06 - March 10, 2006.
□ A. Crăciun, *Lazy Thinking Synthesis of a Groebner Bases Algorithm.* Poster at Calculemus 2006, 13th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, July 7-8-9, 2006, DIMA, Genova, Italy.
□ A. Crăciun, M. Hodorog. *The QuotientRemainder Theorem for Natural Numbers: Discovery by Lazy Thinking.* Talk at First Central and Eastern European Conference on Computer Algebra and Dynamic Geometry Systems in Mathematics Education (CADGME), University of Pcs, Pollack Mihly Faculty of Engineering, Hungary, 20-23 June, 2007.