An Architecture for Combining Provers and its Applications in the *Theorema* System

Dissertation

zur Erlangung des akademischen Grades
"Doktor der technischen Wissenschaften"

Eingereicht von

Elena Tomuţa

Juli 1998

Erster Begutachter: o.Univ.-Prof. DDr. Bruno Buchberger
Zweiter Begutachter: o.Univ.-Prof. Dr. Jochen Pfalzgraf

Angefertigt am: Forschungsinstitut für symbolisches Rechnen
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eidesstattliche Erklärung

Ich versichere, daß ich die Dissertation selbständig verfaßt habe, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch sonst keiner unerlaubten Hilfe bedient habe.

Linz, am 18. Juli 1998 Elena Tomuţa

Zusammenfassung

Diese Dissertation geht aus der Arbeit des Autors im *Theorema*-Projekt hervor. *Theorema* verfolgt das Ziel, ein System zu errichten, das Computer-Algebra und Logik integriert. Der eine Teil des Systems ist ein gemischt interaktiver - automatischer Beweiser, der Mathematikern Unterstützung anbietet, indem er “leichte” Teile von mathematischen Beweisen automatisch erzeugt. Beim Erzeugen von Beweisen imitiert das System die Arbeitsweise vom Mathematikern.

In diesem Rahmen werden zuerst jene Probleme, die beim Beweisen mit Funktoren auftreten, untersucht. Dazu wurde eine einfach Sprache entwickelt und implementiert, die es dem Benutzer erlaubt, Funktoren und Bereiche sowie Beziehungen zwischen ihnen zu definieren. Es wurde ein Funktorbeweiser entwickelt, der mit den Konstrukte der Funktorensprache arbeitet. Weiters wurden Experimente mit konkreten, automatisch erzeugten Beweisen durchgeführt. Der Funktorbeweiser besteht aus einer Kombination von Basisbeweisern in *Theorema*, die jeder einen bestimmten Typ von Beweisschritten beherrscht: Prädikatenlogik höherer Ordnung, Vereinfachung bezüglich einer Menge von Gleichungen, Induktion.

Während der Implementierung des Beweisers und der Experimente traten mehrere Probleme auf, die mit der Art und Weise, wie die Basisbeweiser zusammenarbeiten und wie diese Interaktion an das jeweilige Problem, das untersucht wird, einfach angepaßt werden kann, zu tun hatten. Zur Interaktion der Beweiser wurde ein Verfahren entwickelt und implementiert, das eine Sprache zur Spezifikation von Strategien und Interaktionsmuster während der Suche nach dem Beweis enthält. Dieses Verfahren wird nun in *Theorema* auch außerhalb des Funktorbeweisers verwendet. Es basiert auf bekannten Datenstrukturen und Algorithmen der künstlichen Intelligenz wie UND-ODER Graphen, hat jedoch auch mehrere neue Merkmale, die durch

die besonderen Anforderungen von *Theorema* begründet sind.

Die Dissertation ist praxisorientiert und praktisch alle beschriebenen Sprachkonstrukte und Algorithmen sind in *Theorema* implementiert.

Abstract

This thesis is the result of the author's work in the *Theorema* project whose aim is to build a system integrating Computer Algebra and Logic. One part of the system is a mixed interactive-automated theorem prover which offers support to mathematicians by generating automatically the "easy" parts of mathematical proofs. When generating proofs the system imitates the way in which mathematicians work.

Within this frame, we first investigated the problems appearing when proving with functors. In order to do this we designed and implemented a simple language allowing the user to define functors and domains and to state properties involving them. We designed a functor prover which handles the constructs of the functor language and we carried out experiments with concrete proofs generated automatically. The functor prover is based on a combination of *basic provers* in *Theorema*, each carrying out a special type of reasoning step: higher order predicate logic, simplification with respect to a set of equalities, induction.

During the implementation of the prover and our experiments, several problems appeared related to the way the basic provers interact and to how this interaction can be adjusted easily to the type of problem at hand. We designed and implemented a mechanism for provers' interaction which includes a language for specifying strategies and interaction patterns during proof search. This mechanism is now being used in *Theorema* also outside the functor prover. It is based on known artificial intelligence data structures and algorithms like AND-OR graphs but has several new features which were motivated by the particular requirements of *Theorema*.

The thesis is practice-oriented and virtually all of the language constructs and algorithms described are implemented in *Theorema*.

Acknowledgements

I thank my advisor Bruno Buchberger for starting the *Theorema* project and thus making this thesis possible, as well as for sharing his experience in research work and life. I also thank him for the efforts he made for the creation and growth of RISC-Linz which provided me with a wonderfull environment for research. I thank Jochen Pfalzgraf for agreeing to co-referee my thesis and for his helpful comments. I am indebted to Tudor Jebelean with whom I had extensive discussions about this thesis and whose comments and critique helped me clarify several ideas. I thank the members of the *Theorema* group, Klaus Aigner, Claudio Dupre, Franz Kriftner, Mircea Marin, Koji Nakagawa, Ovidiu Podisor, Yaroslav Uscenko, Daniela Vasaru and Wolfgang Windsteiger, for providing me with useful comments during all stages of my work and for being great friends and colleagues.

Contents

Chapter 1

Introduction

The *Theorema* system ([Buchberger, 1993], [Buchberger, 1996a], [Buchberger, 1996b], [Buchberger, 1996c], [Buchberger, 1996d], [Buchberger *et al.*, 1997a], [Buchberger *et al.*, 1997b], [Buchberger *et al.*, 1997c]) aims at integrating proving support into a computer algebra system. The emphasis is on supporting the trivial parts of proofs while enabling the user to give hints to the system by means of a user-friendly interface. Since the primary motivation of the system is to provide assistance in teaching mathematics and in mathematical research, it is of paramount importance that the language in which users interact with the system is as close as possible to usual mathematical notation. Also, the system should support proving in traditional mathematical areas (natural numbers, abstract algebraic structures, real numbers, etc.)

Very powerful proof systems have been developed so far, with applications ranging from program or circuit verification to helping prove open problems in mathematics. However to our knowledge, no theorem proving system is in use in mathematical education, mainly because of the lack of an appropriate mathematical language. Most systems which satisfy this language constraint are restricted to few fields of mathematics while others, based on powerful type systems, can support work in different domains but require the user to be familiar with advanced type-theoretical in order to be able to effectively interact with the system.

In Chapter 2 we survey existing theorem provers to support the statements above. We outline similarities between other theorem proving systems and

Theorema and review systems for which an integration with *Theorema* might be of interest.

Mathematics in our system is organized by functors [Buchberger, 1996c]. Some primitive mathematical domains are provided a priori: integer and real numbers, tuples, sets. On top of these new domains can be built by functors.

A significant amount of research has been done in the frame of the *Theorema* project on proving in particular areas of mathematics (induction over lists and natural numbers, proofs in analysis, proofs in algebraic structures). These proofs are performed by a couple of *basic* provers: a predicate logic (natural deduction style) prover, a simplification prover, an induction prover, a functor prover. The philosophy of the system is to provide, for each mathematical domain, a specialized prover which incorporates proving techniques characteristic for that domain. Such specialized provers, however, typically consist in a mixture of several kinds of deduction steps: predicate logic steps, simplification steps, application of definitions, all combined along some domain-specific interaction pattern.

In earlier versions of the system, domain-specific provers, were stand-alone provers and re-implemented the parts of the basic provers they needed. In this approach the development of a new domain-specific prover is a lengthy operation in which the developer is forced to cope with implementation details of the basic provers involved rather than focusing on the study of the proof-techniques of the domain she is working in. Another attempt at implementing a mechanism for supporting provers interaction was undertaken in the functor prover (see Chapter 4). Here the predicate logic prover, the simplification prover and some functor-specific deduction steps were combined into one prover. In this approach, although no parts of the basic provers were rewritten, the implementation of the interaction mechanism itself becomes cluttered because of a mixture of operational and heuristic elements. It thus became clear that a mechanism for proof-search and provers interaction is necessary which allows the developers of *Theorema* to quickly implement new domain-specific provers, without having to rewrite or duplicate parts of the existing basic provers and such that a particular interaction pattern for the basic provers involved can be implemented easily. After several experiments with mathematical proofs in earlier versions of the *Theorema*, we implemented a system which searches for a proof in a tree-like structure representing the proof-object. Basic provers participating in the proof as well as constraints describing the way in which these provers interact can be specified by means of a simple proof strategy language. Thus, domain-specific proving translates into designing sets of proof strategy specifications for each mathematical domain. The code of the basic provers themselves remains unchanged.

The mechanisms for provers interaction and the proof strategy specification language are presented in Chapter 3.

Our interest in provers interaction problems was motivated by some experiments with proving *knowledge-transport properties*, i.e. properties which, if present in some mathematical domains carry over into a new domain built from the given ones by means of a functor. In such proofs, typically, one tries to reduce proving a property in the "target" domain to proving one or more properties in in the underlying domains. More specifically, we focused on proving algebraic properties of domains built up by functors which are used in constructing new number domains from existing ones (complex numbers, rationals, integers).

In order to be able to formulate these properties we developed a simple functor language extending the *Theorema* language, presented in Chapter 4. Inference rules on these new language constructs are implemented in a functor prover described in the same chapter.

The results of our case study concerning knowledge transport properties are in Chapter 5. Although the functor prover is not yet integrated with the proof-search mechanism in Chapter 3 we show what proof-strategy specifications can be used for proving the knowledge-transport properties in Chapter 5 and also how integration with the new proof-search mechanism would enlarge the set of theorems our system is able to prove automatically.

Chapter 2

Existing Theorem Proving Systems

In this chapter we review some of the many theorem provers developed so far. The list is by no means complete. We have selected some systems which we judged to be among the most prominent ones and which are relevant to the following chapters of this thesis for one of the reasons listed below:

- For automated theorem proving systems we compare their proof-search mechanism with the one of *Theorema* presented in Chapter 3.

- Theorem provers based on type-theories are of interest because *Theorema* aims to organize mathematics by functors, which are similar to type constructors. All systems we encountered were, however, unlike *Theorema*, based on some constructive logic. This has deep consequences on the style of proving and thus the proofs involving functors in *Theorema*, given in Chapter 5 are quite different from the ones generated by theorem provers based on constructive logic.

- Finally we review some state-of-the-art systems performing resolution theorem proving or applying symbolic methods to theorem proving since such modules could be called by *Theorema* to perform certain parts of proofs in special domains.

Although developed almost 20 years ago, the Boyer-Moore theorem prover [Boyer, Moore, 1979] is still cited among the state-of-the-art induction theorem provers. The system offers support in program verification and is thus centered on proving theorems about inductively defined objects. Boyer and Moore developed a formal theory in which they construct their proofs and which is tailored to the needs of reasoning about computer programs. Central concepts of the theory are: the cell principle, through which new inductive "types" can be defined; the definition principle allowing the definition of new recursive functions; a generalization of the Noetherian induction principle which is then customized for the needs of individual theorems to give theorem-specific induction schemas. When defining new types of objects, a bottom object and a well-founded relation are provided so that any object of the defined type can be constructed from the bottom object by applying the constructors of the object a finite number of times. When defining new functions the user is asked to provide a well-founded relation which is used in proving that a unique function exists which satisfies the given definition.

The prover applies rules of the propositional calculus with variables and function symbols, equality reasoning, instantiation and induction.

Powerful heuristics are provided for building the induction scheme used in each particular case as well as for solving the induction base case and step case. The conjectures to be proven are represented as clauses. According to a waterfall metaphor, a fixed set of heuristics are tried out in order on the initial clause resulting in a new set of clauses whose conjunction implies the initial conjecture. Proven clauses are removed from the set. The remaining clauses are then, in turn passed through the waterfall and the process is repeated until no more clauses remain to be proven.

Interestingly enough, although there is virtually no proof planning mechanism employed, but merely a fixed set of heuristics tried successively on a sequent, the set of theorems to be proved, including the unique prime factorization theorem represent even today a benchmark for inductive theorem provers.

The reason probably lies (at least in part) in the very careful l consideration of the proving techniques used in inductive proofs. In *Theorema* the setting is slightly more complicated due to the large number of mathematical domains (and specific proof techniques) we aim to support.

There were several attempts to reorganize and systematize the heuristics of the Boyer-Moore prover, e.g. Bundy ([Bundy *et al.*, 1992]) expressed them by using *rippling* as a unifying strategy, however, the improvements in terms of theorems proved were not dramatic.

The typical use of the Boyer-Moore prover is in proving the correctness of computer programs.

Apart from *Theorema* the only other prover implemented on top of Mathematica that we are aware of is the Analytica prover developed at the Carnegie Mellon University by E. Clarke and X. Zhao (see [Clarke, Zhao, 1993]). Analytica makes strong use of the symbolic computation facilities provided by Mathematica and is to our knowledge the theorem prover which exploits symbolic computation facilities best. For instance, Analytica was used to prove completely automatically several combinatorial equalities ([Clarke, Zhao, 1994]) taken from the first chapter of Ramanujan's Notebooks ([Berndt, 1985]) by using symbolic summation techniques like the Gosper's algorithm combined with heuristic steps. Other supported areas are proofs in analysis and induction proofs. The analysis prover includes a generalized version of the SUP-INF method for solving linear inequalities, which also handles a large class of non-linear inequalities.

Analytica attempts to apply some built-in heuristics to each formula it is asked to prove. A formula to be proven passes through several phases like simplification by a some logical and algebraic rules, attempt to apply some of the existing inference rules and rewriting to a simpler, equivalent form. Hence a fixed list of transformations is applied, with no search involved.

Oyster-Clam [Horn, Smaill, 1990], [Bundy *et al.*, 1990] consists in an interactive proof editor (Oyster) based on the Cornell NuPRL system but implemented in Prolog, and the meta-level proof-planning system Clam [van Harmelen, 1989] built on top of Oyster enabling the user to do fully automatic proofs. The metalanguage of Oyster is Prolog, the object language is a version of Martin-Löf type-theory in a sequent calculus formulation. A set of tactics and tacticals is provided in Oyster for use in proving. Clam adopts Oyster tactics by adding a specification for each tactic. The resulting methods ([Bundy, 1988]) consist in an input formula, preconditions, output formulae and postconditions. Methods are used while automatically building a proof in a forward search process (i.e. applicable methods are determined for the current sequent, then the schematic output and the postconditions are computed for the respective goal. The process is repeated on the formulae obtained until no more sequents remain to be proven. The methods implemented are modeled on a rational reconstruction of the Boyer-Moore theorem prover and include the rippling technique introduced by Bundy [Bundy *et al.*, 1992]. Being based on a higher order constructive logic including induction the system supports the extraction of algorithms from proofs, in the form of terms in an enlarged λ-calculs. A successor of Oyster, which is completely independent of the logic used, called Mollusc, [Richards *et al.*, 1994] was developed in 1992.

An interesting crossing between theorem proving and expert system techniques is MUSCADET [Pastre, 1989]. The system employs rules and meta-rules which are selected and applied by an inference engine. At each moment during the search process a set of active rules is available. Active rules and their order can change during a proof as a result of the application of a meta-rule.

The inference engine selects the applicable rules and applies all of them in all possible ways. The process continues until either no more rules can be applied or until one of the rules claims success. Some meta-rules generate rules automatically from declarative knowledge. For instance for each definition several rules (which may be used to apply the definition to a term) are added to the rule base. This is also how the functor module described in Chapter 4 handles domain operator definitions. Similarly, some lemmas to be applied in the proof are automatically transformed into rules and not used as formulae. Heuristic knowledge is implemented for particular sets of problems by means of meta-rules. These meta-rules specify that if the hypotheses and conclusion are of a certain form (sub-terms *match* a certain given expression) then certain rules should be applied or certain transformations of the rule base should be undertaken.

The corresponding construct in *Theorema* is the proof-strategy specification language, which allows the developer to specify in what conditions a certain basic prover should be applied to a proof situation during proof search. We also note that while MUSCADET implements its rules and proof search engine in PASCAL, we used *Mathematica* whose functional nature and advanced pattern-matching facilities made the proof-strategy specification language fit naturally into the existing programming language of *Mathematica*. The notion of meta-rules in MUSCADET, i.e. of transformations applied during a proof to the parameters defining the heuristics of the system also carries over nicely to *Theorema*: It is possible to implement some special "basic provers" complying to the same formal specifications as "usual" basic provers (natural deduction, simplification, induction provers) but whose actions consist in modifying the current set of proof-strategy specifications based on the existing ones and on the deduction steps already performed. Such "meta-provers" have not yet been implemented but they will undoubtedly be needed in more complex proofs and can be integrated naturally in the current architecture of our system.

MUSCADET was written in Pascal and was used to prove theorems in set theory, mappings, relations, topology and topological linear spaces. More recently its application to discrete geometry [Pastre, 1993]of cellular automata was studied.

One aspect of MUSCADET we particularly liked was the effort put into imitating human proof techniques, which goes beyond that of most systems we know. The author observed and recorded the actions taken by three mathematicians when presented with a non-trivial theorem in topology ([Pastre, 1989]) and modeled their attempts and analogies by means of meta-rules.

A system similar in its goal with *Theorema* is Omega [Benzmüller *et al.*, 1997] whose aim is to support theorem proving in main-stream mathematics and mathematics education. The Omega system contains a proof-planner based on an extended STRIPS [Fikes, Nilsson, 1971] algorithm. Furthermore, like *Theorema*, Omega aims at integrating computer algebra support into proving. However, this integration is based on the requirement that the computer algebra system used produces high-level protocol information from which proof-plans can be derived automatically. The proof-search engine in Omega uses *methods* which fully specify the input/output behavior of the tactics to which they are associated, to the point where it is possible to automatically generate the tactic from its specification (method). This is a very elegant approach and it has the advantage that the proof-planning mechanism may manipulate methods (i.e. adapting general ones to special mathematical domains) and the corresponding tactics will then be automatically generated from the methods thus "synthesized". However, the situation in *Theorema* is quite different: basic provers have a heuristic of their own, which can be modified in order to enable them to handle specific classes of proofs. Such provers can use *Mathematica* procedures in order to simplify terms or may even call an external prover for performing some special subproof. We think that it is unrealistic at least in the short term to aim to decompose these provers into elementary, totally specified building blocks. Rather, the view adopted in this thesis is that such basic provers are *agents* performing parts of a proof which "they know how to handle".

McAllester's system Ontic [McAllester, 1987] is a proof checker based on Zermelo-Fraenkel set theory augmented with the notions of *type* and *function*. One distinguishing feature of Ontic is the inference mechanism it uses. Unlike most proof checkers using backward chaining, rewriting or unification as basic inference mechanisms, Ontic's inference can be:

congruence closure, i.e. deriving $P(f(b,c))$ from $P(f(a,d))$ if $a = b$ and $c = d$ are known;

boolean constraint propagation which propagates truth values so that for instance, from P and $\neg P \cup Q \cup R$, $Q \cup R$ is derived;

focused binding which roughly means instantiating universally quantified variables in a formula with proper constants, using congruence closure and type constraints;

universal generalization which allows the system to derive $\forall x \phi(x)$ if $\phi(c)$ is known for a constant c about which no additional assumptions are made.

Although Ontic is a proof checker, the user does not overtly supply proof steps nor name the facts to be used, but guides the system towards the proof by highlighting certain focus objects and providing suppositions. To achieve this the axioms and schemes of set theory are built into the syntax so they can be better exploited by Ontic's inference mechanism and the user does not need to invoke them explicitly in proofs. The result is a system which tackles the problem of getting computers to check mathematical proofs by verifying details in *a natural argument* provided by a human user, an aim similar to the one of *Theorema*. Ontic, however, was applied mostly in proofs in set theory and boolean algebra (a highlight is the verification of the Stone representation theorem) and does not adequately support proofs involving induction, arithmetic, inequality reasoning and rewriting. The proof strategy language defined in Chapter 3 is hopefully appropriate for expressing natural arguments provided by *Theorema* users and/or developers.

Among the theorem proving systems based on type theories and aiming at formalizing mathematics we noted NuPRL [Constable, 1986] and Coq [Dowek *et al.*, 1991]. NuPRL is based on Martin-Löf's type theory [Martin-Löf, 1982]. Being based on a constructive type-theory NuPRL exploits the notion of 'propositions as types' or 'proofs as programs'. A logical expression corresponds to a type, with the type being inhabited only if the predicate-logic expression is provable. The logical expression can then be viewed as the specification of the associated term.

Jackson [Jackson, 1995] studied how abstract algebra can be implemented in NuPRL, developing notions such as groups of permutations of arbitrary types and finite sets, finite multi-sets, a theory of summation over groups, monoids, rings, modules, algebras, polynomial arithmetic. Doing mathematics in a constructive setting requires, however, a good understanding of constructive logic and of the limitations it poses in defining and manipulating mathematical notions. Systems based on constructive logic may be very well suited for reasoning about computation because they provide as primitive notions ways of constructing primitive recursive functions. However doing mathematics in such a system is most of the time quite different from the mathematics one reads in textbooks.

The system Coq is based on the Calculus of Constructions [Coquand, Huet, 1988] and supports extraction of a certified program from the constructive proof of its formal specification, the focus thus being automated synthesis of correct programs from formal specifications.

Both Coq and NuPRL offer proof development support by means of tactics which can be applied to sequents under development but do not prove theorems automatically.

A special position among the systems intended to assist mathematicians is that of Mizar [Trybulec, Blair, 1985], [Rudnicki, 1992]. Mizar is a proof-checker based on "the composite system of logic" of Stanislav Jaskowski [Jaskowski, 1934] and on the Tarski-Grothendieck axiomatization of set theory. The idea is to offer mathematicians the possibility to write papers by defining new concepts and proving theorems about them in a strictly formalized manner with each inference step in a proof being checked by the system. In this process notions and results from existing Mizar articles can be used by a feature called cross-referencing. According to [Rudnicki, 1992] it is this cross-referencing feature together with a sophisticated input language that made Mizar into a comparatively widely used system. Through contributions of mathematicians a database of mathematical articles was built, which in 1992 contained about 300 entries. The articles are published periodically in "Formalized Mathematics", by the Universite Catholique de Louvain. The Mizar database is hence (as far as we know) the largest corpus of formalized and organized mathematical knowledge in electronic form.

Although the input to Mizar is in text mode, the language seems to be close enough to the one used by mathematicians such that they can understand it and learn it relatively easily. Mizar articles are automatically typeset in TEX. The papers in the Mizar database are mostly formalizations of known results and contain few new ones.

Many interactive theorem provers based on tactics have been developed, some of the best known being: HOL, [Gordon, 1987] and its predecessor LCF [Gordon *et al.*, 1979], Isabelle [Nipkow, Paulson, 1992], IMPS [Farmer *et al.*, 1993], the Larch Prover [Garland, Guttag, 1991].

HOL is a natural deduction proof system based on classical higher order logic with Milner-style polymorphism, which uses ML as a meta-language for defining tactics and tacticals. Isabelle is a generic theorem prover in which logics can be specified by giving their syntax and rules of inference. The supplied tactics include a simplifier able to handle conditional and permutative rewrite rules, perform automatic case-splits and extract rewrite rules from contexts. The logic of IMPS is a non-constructive version of simple type theory with partial functions and subtypes. Proofs in the system are obtained by successively applying proof commands to a deduction graph (a data structure representing the proof).

In interactive proof systems the user has to explicitly tell the system what tactic to apply at each point in the proof. However, the gap between widely used interactive proof systems and automated ones decreases as users of interactive proof systems provide more and more sophisticated tactics able to perform many deduction steps in one batch or applicable to specialized fields of mathematics. Thus, such systems start as interactive provers and evolve towards automated ones. *Theorema* simply starts from the other end of the spectrum by offering the user functions which attempt to automatically generate a proof of a given conjecture, then integrating more and more possibilities for user control of the search process and heuristics employed. Ultimately, the starting point is determined by the intended initial audience of each system. *Theorema* is intended to be used by students receiving training in mathematical proving, who have no knowledge of the field of artificial intelligence. Such users should be able to use the system for getting a proof or at least a proof attempt for a given conjecture even without knowing how the system works internally. Better knowledge of the system will enable them to control the proof-search mechanism better and ultimately prove more theorems. Eventually, for very complex proofs, *Theorema* will probably need to be guided through the proof like an interactive theorem prover.

Finally, we would like to list some theorem provers which support proving in few fields of mathematics only but implement state-of-the-art techniques for automated theorem proving techniques in these fields. An interface between these provers and *Theorema* might be of interest.

In the area of resolution theorem provers, one of the most impressive systems we know is OTTER (Organized Techniques for Theorem Proving and Effective Research) [Wos, McCune, 1992], a system which was successfully used for answering some open questions in Mathematics and Computer Science. The program accepts input in the language of clauses or in first order predicate logic with equality. It uses various flavors of resolution (binary resolution, hyper-resolution, UR-resolution), subsumption and paramodulation as inference rules. The proof search can be guided by strategies such as the set of support strategy and the weighting strategy. Clauses can be designated as demodulators, in which case they will be used for rewriting expressions to a canonical form. Typically OTTER proofs are proofs by refutation but they are extended by providing a literal called ANSWER through which the user has access to the object that may be implicitly found when a refutation proof succeeds. For instance, in order to prove a statement of the form $(\exists x)P(x)$, OTTER will negate the conclusion and try to deduce the empty clause. Upon proof completion, the ANSWER literal will hold a witness term for predicate P Such was the case when OTTER succeeded to prove fixed point properties in combinatory logic or existence of combinators satisfying certain equations.

In equivalential calculus OTTER was used for proving (by refutation) that certain formulae are single axioms for the equivalential calculus. In the quest for proving that every Robbins algebra is a boolean algebra, OTTER was used to help prove some sufficiency properties (i.e. properties which, if added to the axioms of a Robbins algebra guarantee that it is a boolean algebra). The system can also be used for computational tasks like finding the closure of a set of generators which additionally satisfy given equations.

To summarize, OTTER offers remarkable assistance in solving problems which lend themselves to translation into the language of clauses and do not involve a very large number of clauses. Guiding the system to produce a proof requires deep insight into the respective research area but also into the way the system uses strategies and inference rules. Hence OTTER's audience consists mainly in users with advanced knowledge in resolution theorem proving. On the other hand finding a complicated witness term for a given property is a subproblem which occurs quite often in mathematical proofs so it would be interesting to study the possibility of calling OTTER from *Theorema*.

The RRL (Rewrite Rule Laboratory) prover [Kapur, Zhang, 1989] is based on rewriting and provides facilities for proving theorems in first order logic with equality, generating decision procedures for first-order equational theories using completion procedures and proving theorems by induction (through methods based on the proof by consistency approach as well as through direct inductive proof).

RRL takes as input a finite set of axioms and transforms each of them into an equation or into a conditional equation. For first order formulae, this is achieved by decomposing formulae into smaller ones by the use of properties of the boolean connectives, applying skolemization and then forming equations using exclusive-or and and connectives.

The core of the system is the extended Knuth-Bendix completion procedure [Knuth, Bendix, 1970] which is applied in various instances: theorems in first order logic are typically proved by refutation and the Knuth-Bendix completion is used to prove inconsistency of a system of equations. Proofs by induction by the consistency check method are reduced to a similar application of the Knuth-Bendix completion procedure.

The RRL system provides the user with implementations of the state-of-the-art rewriting methods. As in OTTER's case, in order to make full use of the power of the system the user must have a substantial knowledge of the inner workings of the system. For instance, in the phase of transforming equations into rules, the user may specify the order on function symbols to use; significant fine-tuning is possible at the level of the completion procedure where the strategy for processing the next equation as well as the critical pair strategy can be set by the user. However, as with OTTER, it would be of interest to study the possibility of linking RRL to *Theorema* through the basic prover construct we present in Chapter 3.

Chapter 3

A Mechanism for Combining Provers

3.1 Terminology

The object language in *Theorema* is a version of higher order language without extensionality([Buchberger *et al.*, 1997b], [Buchberger, Kriftner, 1997]). We denote this language with ThO.

The metalanguage is the programming language of the *Mathematica* system.

For a data type D, we denote by $Lists(D)$ the data type of lists of elements in D. An element in $Lists(D)$ is the empty list or is of the form $\{d_1, \ldots, d_k\}$ with d_i in D for $i = \overline{1, k}$.

The proof search mechanism we present in this chapter makes no use of an underlying object language and thus is completely general.

We will only refer in this chapter to valid formulae formulae F of ThO and lists of such formulae, i.e. members of the data type $Lists(ThO)$.

A *proof situation* in *Theorema* is a pair (g, a) consisting in a goal $g \in ThO$ and a list of assumptions $a \in Lists(ThO)$.

We denote by PV be the data type consisting in the identifiers $proved$, $failed$, $pending$.

3.2 Introduction

Building up mathematics by functors has the natural consequence that provers for various areas of mathematics can be structured in a similar way. The idea is thus to associate special provers with functors such that the knowledge about algebraic structure which is contained in functors can be used for reducing the complexity of proof-search. See also [Buchberger, 1993] for a discussion on the gap between computer algebra systems and mathematical assistants.

In practice the special provers corresponding to various functors will need to support various kinds of reasoning steps like natural deduction, handling of equality, case distinctions, applying definitions, induction and computation. Therefore a mechanism for *combining provers* is of particular importance for the development of the system.

The problem we want to solve is the following:

Given a set of provers P_1, ..., P_n, each implemented as a set of rewrite rules and each applicable to a some schemata of goal and assumptions list

Design a mechanism enabling the provers P_1, ..., P_n to cooperate in finding the proof of a formula under given assumptions, by performing parts of the global proof such that:

- The mechanism is independent of the code of the prover involved.
- The mechanism allows easy specification of additional conditions on which provers are called at particular points in the proof.
- Fine-tuning the interaction mechanism, addition and removal of provers does not involve changing the implementation of the provers or of the interaction mechanism.

The provers P_1, ..., P_n are called *basic* provers.

Our solution to the problem, presented in this chapter consists in several components:

- the description of the global data structure used for finding a proof and for producing a natural-language representation thereof (in the sequel we refer to this data structure as *global proof-object*);

- the specification of the conditions a *Mathematica* function must fulfill in order to qualify as *basic prover*;
- an algorithm for searching for a proof in the global deduction tree while calling the basic provers available;
- a proof strategy specification language which can be used to specify restrictive conditions on the steps in the proof at which selected basic provers are called.

The mechanism for combining provers which we describe *searches for a proof* in the space of all possible deductions starting from a given proof situation.

Our work has some similarity with (and was influenced by) research on proof planning in automated theorem proving. The global deduction tree data structure is similar to the AND/OR graphs introduced by Nilsson (see [Nilsson, 1980]) but unlike those, the structure we use is a tree (not a graph) and may contain a third type of nodes, the IF-nodes. Bundy ([Bundy, 1988]) introduced the notion of method by extending a tactic with its specification which can be used by a planner; the Oyster-Clam system ([Bundy *et al.*, 1990]) implements these concepts. [Huang *et al.*, 1994] proposes a different notion of a method, in which the list of logical inference rules applied by the tactic are declared as part of the method definition. The procedural part of the tactic is then generated automatically by means of an interpreter from the declarative part. In this way all knowledge about the logical operations performed by the method is contained in the declarative part making it easier to automatically modify the method.

Our approach is similar because basic provers can be seen as methods (or plan operators) and the mechanism for combining provers plays the role of a planner which selects and "applies" basic provers in order to find a proof.

The difference from the approach taken in planning consists in the fact that we see special provers as encapsulating both proof control heuristics and operational content. This view follows from the aim of *Theorema* to unify in one system specialized proof engines based on algebraic techniques, like the Knuth-Bendix procedure for proving in equational logic, Collins' quantifier elimination method ([Caviness, Johnson, 1998]) or the Groebner Bases and characteristic set methods used in proving some classes of theorems in elementary geometry ([Buchberger, 1993]). Such provers modules have built-in heuristics which we would like to be able to integrate as fully as possible into the global proof search mechanism. Clearly, an existing prover incorporated into *Theorema* will need to be wrapped in a software module performing the necessary translations between the data structures of the respective prover and the ones of *Theorema*.

Although we search for a proof, the intended use of the system is not in searching huge spaces of possible deductions. Many successful automated theorem provers (e.g. Nqthm [Boyer, Moore, 1979], Analytica [Clarke, Zhao, 1993]) achieved remarkable results by applying a carefully designed fixed set of heuristics to problems in a certain field of mathematics. We want to offer the developer of domain specific provers in *Theorema* the possibility to quickly specify such a set of heuristics, thus reducing drastically the search space. The possibility of search in a limited space of deduction is however, a useful feature also in the stage of designing and refining domain specific heuristics.

Last but not least, *Theorema* aims to be a mixed interactive-automated theorem prover. It should be possible to carry out the trivial parts of proofs automatically and ask for assistance for solving the ingenious parts. This means that the user interaction mechanism should be smoothly integrated with the global proof search module.

We proceed to describe the components of the mechanism for combining provers and conclude the chapter with an example illustrating the use of the concepts presented and a discussion of the support for interactive proving.

3.3 Deduction Trees

In automated theorem proving several formalisms have been used for modeling the process of proving. In the interactive proving community, one approach represents proof states as sequents which by applying tactics are transformed into a set of sequents such that if each of the resulting sequents is proved then also the initial one is. In the proof planning community proving means finding a sequence of plan operators, which, when applied to the state representing the initial theorem result in a state with no open subgoals. A common representation for proof states in this case is the STRIPS notation (see [Nilsson, 1980]).

In *Theorema* the data structure used to represent a stage in the proof of a formula is the *deduction tree*.

As in tactic-style theorem proving we reduce a given proof situation to several new, "simpler" proof situations. The reduction is such that if all resulting proof situations are proved, also the initial one is proved.

Given a proof situation there may be several ways in which it can be reduced to new proof situations (i.e. we may have several methods at our disposal for proving the theorem) and it is possible that while searching for a proof we must examine several or all such alternatives to find the "right" reduction.

Consequently, as in [Nilsson, 1980] deduction trees in *Theorema* may contain OR nodes corresponding to the various (disjunctive) alternatives in reducing a proof situation to new ones and AND nodes corresponding to the (conjunctive) composition of subproblems to which a given one is reduced. Furthermore, during proving, it may be the case that the outcome of a deduction step depends on whether another formula is proved under the current set of assumptions. To accommodate this type of situation, we have also introduced IF nodes, which model the situation in which a proof situation is reduced to a new proof situation depending on whether a certain condition (which must also be proved) turns out to be true. IF nodes were introduced mainly in order to handle case definitions or definitions of partial operators with which *Theorema* basic provers deal.

Clearly, it is possible to employ a representation of the proof by using only AND nodes. OR nodes and IF nodes reflect proof control elements which could be hard-coded in the mechanism for combining provers itself. However, as stated before, the basic provers which build up the proof of a formula have a built-in heuristics themselves. By using OR and IF nodes in their output they can impact on the way the proof search is done.

Deduction trees in *Theorema* serve a double purpose:

- they record the transformation steps performed during a proof from which a natural language representation of the proof is generated.
- they serve as the data structure upon which the search for proofs in the space of all possible state-reductions is performed.

A deduction tree *DT* is a tree in which each node is either:

- an open proof situation;
- a non-terminal deduction step node, i.e. an AND node, an OR node or an IF node;

- a terminal node i.e. a node which represents the final step in a proof, resulting in either a failed or proved proof situation.

Non-terminal deduction step nodes and terminal nodes have as components:

- a trace of the deduction steps performed, used for displaying a natural language representation of the proof,
- the current proof situation on which the respective deduction step is being applied,
- a list of successor nodes, which is empty if and only if the respective node is terminal.
- a *proof value* which can be one of *proved*, *pending*, *failed.*

In particular IF nodes have exactly 2 successor nodes.

Each subtree in the deduction tree represents the attempts made to prove the goal formula of the proof situation at the root of the subtree.

3.3.1 Deduction Trees: A Formalization

We now formalize the notion of deduction tree in order to be able to describe more clearly what operations on proof situations are performed during proof search.

In the formalization we do not mention the component of the deduction tree nodes which is not relevant for proof search: the trace of the deduction step performed.

In the remainder of this section we will denote by $Nodes(T)$ the set of nodes of the tree T.

For the few notions of graph theory appearing in this presentation we followed [Williamson, 1985].

Definition 1 *A* deduction tree *is a tree with nodes labeled with triples* (t, g, a) *where*

t is one of the identifiers AND, OR, IF, PS (henceforth called the type of the node)

$g \in ThO$ *is called the* goal of the node,

$a \in Lists(ThO)$ *is called the* list of assumptions of the node.

In addition nodes of type IF have exactly 2 successors. Nodes of type PS have no successors.

Definition 2 *For a given a deduction tree T, a function $f : Nodes(T) \to PV$ is called an* annotation *of T.*

In order to be able to express the following concepts easier, we define a function $\Pi : \{AND, IF, OR, PS\} \times Lists(PV) \to PV$ in the following way:

$$\Pi(AND, \{p_1, \ldots, p_k\}) = \begin{cases} proved & \text{if } \forall i \in \overline{1,k},\ p_i = proved \\ failed & \text{if } \exists i \in \overline{1,k},\ p_i = failed \\ pending & \text{otherwise} \end{cases} \quad (3.1)$$

$$\Pi(OR, \{p_1, \ldots, p_k\}) = \begin{cases} proved & \text{if } \exists i \in \overline{1,k},\ p_i = proved \\ failed & \text{if } \forall i \in \overline{1,k},\ p_i = failed \\ pending & \text{otherwise} \end{cases} \quad (3.2)$$

$$\Pi(IF, \{p_1, p_2\}) = \begin{cases} p_1 & \text{if } p_1 \neq proved \\ p_2 & \text{otherwise} \end{cases} \quad (3.3)$$

Note that Π is a partial function. It is not defined e.g. on pairs of the form $IF, \{p_1, p_2, p_3, p_4\}$ where p_1, p_2, p_3, p_4 are in PV.

Remark 1 *If $\Pi(t, \{p_1, \ldots, p_k\}) \neq pending$ then there exists $i \in \overline{1,k}$ such that $p_i \neq pending$.*

This property is easy to check from the definition of Π and we will use it later on in this section.

Definition 3 *For a deduction tree T, an annotation $f : Nodes(T) \to PV$ is called* admissible *if the following conditions hold:*

- *for any terminal node N of T which is not of type PS, $f(N) \neq pending$;*
- *for any node N of type PS, $f(N) = pending$.*

- *for any non-terminal node N with successors $N_1, \ldots, N_k$, and type t, $f(N) = pending$ or $f(N) = \Pi(t, \{f(N_1), \ldots, f(N_k)\})$.*

Intuitively, a deduction tree is syntactically correct if a certain relation (expressed formally through the function Π) holds between the proof value of a node and the proof values of its successors. We also allow deduction trees in which the proof value of the parent node was not yet computed in terms of the proof values of its successors, in which case the proof value of the parent node has to be set to *pending*. The notion of admissible deduction trees models *syntactically correct deduction trees*. We will require later on in this section that basic provers return only such syntactically correct deduction trees.

Lemma 1 *If $f : Nodes(T) \rightarrow PV$ is an admissible annotation for the deduction tree T, then for any non-terminal node $N_0 \in T$, with $f(N_0) \neq pending$, there exists a finite path $N_1, \ldots, N_k$ such that for each $i = \overline{0, k-1}$, N_i is the predecessor of N_{i+1}, $f(N_i) \neq pending$ and N_k is terminal.*

Proof Let N_0^1, ..., N_0^m be the successors of N_0, for some $m \geq 1$. Since T is admissible and $f(N) \neq pending$ it follows that $f(N_0) = \Pi(type(N_0), \{f(N_0^1), \ldots, f(N_0^m)\})$ and further, $\Pi(type(N_0), \{f(N_0^1), \ldots, f(N_0^m)\}) \neq pending$. By Remark 1 for at least one of the successors, (we denote it by N_1) $f(N_1) \neq pending$ holds. If N_1 is non-terminal we apply the same reasoning for N_1 and obtain a node N_2 with $f(N_2) \neq pending$. Since the tree has finite depth, after repeating this construction a finite number of times, we get the path $N_1, \ldots, N_k$ with the desired properties.

Definition 4 *For a deduction tree T, an admissible annotation $f : Nodes(T) \rightarrow V$ is called* correct *if, for any non-terminal node N with successors $N_1, \ldots, N_k$ and type t, $f(N) = \Pi(t, \{f(N_1), \ldots, f(N_k)\})$.*

Intuitively, deduction trees with a correct annotation are deduction trees in which the proof value of each node was computed in terms of the proof values of its successors. We anticipate a bit to explain why such deduction trees are important. The result of the development in this and the following section will be that if the proof value of a node is *proved* then its proof situation is proved. For a node in an admissible derivation tree it could be the the case that its proof situation is proved but "we do not yet know it" because the proof value is not yet computed in terms of the proof values of its successors. That is, the proof value of the node is currently *pending* but after computing by means of the function Π we might get the result *proved*, meaning that the proof situation of the node is proved, or *failed* meaning that we cannot prove the proof situation. A correct deduction tree is a tree for which the proof values were computed for each node in the tree. In the rest of this section we deal with carrying out this computation.

Lemma 2 *For a derivation tree T and an admissible annotation $f : Nodes(T) \to PV$ there exists a unique correct annotation $f' : Nodes(T) \to PV$ such that for any terminal node N of T, $f'(N) = f(N)$.*

A detailed proof of Lemma 2 can be done by induction. It is clear however, that the function f' can be computed in a bottom up fashion from the terminal nodes, since the value of f' in a parent node is a function (Π) of the value of f' in the successor nodes, hence we skip the detailed induction proof.

The lemma also gives an algorithm for computing a correct annotation function. We denote by $depth(N)$ the number of predecessors of the node N and we compute, in each step of the algorithm $f'(N)$ for all nodes N such that $depth(N) = k$ where k is an integer. By letting k run from the depth of the tree to 0 we can compute the value of f' in each node. This crude algorithm is very ineffective since it passes through every node of the tree.

In practice we use the following algorithm:

Algorithm 1

In: a deduction tree T and an admissible annotation f for T.

Out: a correct annotation f' for T such that $f' = f$ on all terminal nodes of T.

1. *let $\mathcal{T}$ be the list of all terminal nodes of T on which the value of f is different from pending.*
2. *let $f' = f$.*
3. *while $\mathcal{T} \neq \{\}$, do*

4. *let N be the first element of $\mathcal{T}$ and remove N from $\mathcal{T}$.*

5. *while $depth(N) > 0$*

6. *Let P be the predecessor of N.*

8. *Let $f'(P) = \Pi(t, \{f'(N_1), \ldots, f'(N_k)\})$ where N_1, ..., N_k are the successors of P and t is the type of P.*

9. *If $f'(P) \neq pending$, let $N = P$, else break.*

10. *endWhile*

11. *endWhile*

Theorem 1 *Algorithm 1 terminates and is correct.*

Proof The algorithm obviously terminates because the number of terminal nodes and the depth of the tree are finite.
It is easy to see that if f is admissible then f' is also admissible, because of the definition of admissible annotations and because it is equal to f on all inputs except on the ones computed by executing line 8 of the algorithm. For the correctness we assume the contrary, namely that the output annotation f' is not correct. Then there exists a node N with successors N_1, ..., N_k, such that the equality

$$f'(N) = \Pi(type(N), \{f'(N_1), \ldots, f'(N_k))\}) \tag{3.4}$$

does not hold. Then from 3.4 and f' admissible it follows that $f'(N) = pending$ and $\Pi(type(N), \{f'(N_1), \ldots, f'(N_k)\}) \neq pending$. Hence by Remark 1 there exists a successor of N, N_m such that $f'(N_m) \neq pending$. Furthermore, $f(N_m) = f'(N_m)$. If this was not the case then the value of $f'(N)$ must have been computed in line 8 of the algorithm and since $f'(N_m) \neq pending$, the computation must have continued by setting $f'(N)$ to satisfy 3.4, which contradicts our assumptions.
From $f(N_m) = f'(N_m)$ and $f'(N_m) \neq pending$ it follows that $f(N_m) \neq pending$ and now since f is admissible, by Lemma 1 there exists a path N'_1, ..., N'_d such that N'_d is terminal, N'_1 is a successor of N_m and for all $i \in \overline{1, d}$, $f(N'_i) \neq pending$. But then it follows that node N_m and hence also node N were considered at some point in line 8 of the algorithm, while defining the value of f' for the predecessors of the terminal node N'_d. Hence 3.4 must hold, which contradicts the assumption we made.

3.3.2 Semantics of Deduction Trees

Until now we have only shown how proof values (i.e. in a formal setting annotations) are manipulated by the proof search mechanism. In this section we make the connection between the handling of proof values in the deduction tree and the inference rules used by the various basic provers participating in the proof search. More specifically, we make the connection between the proof value associated to a node, the type and the proof situation in the node. In the previous section we only talked about types and proof values of a node.

This section has a practical motivation as well: the developer of a basic prover typically starts with a calculus containing the inference rules the respective basic prover will apply. She then has to transform the rules of the calculus plus possibly newly deduced rules into deduction trees to be returned by the prover. This section gives the rules according to which this transformation must be made.

In order to remain general, we base the discussion on a relation $\vdash$ with the following meaning: given a list of assumptions we write $a \vdash g$ if g is a logical consequence of a.

In terms of this relation we can now formulate the following *semantic rules*.

- If $N = (AND, g, a)$ is a node with successors $N_i = (t_i, g_i, a_i)$ for $i \in \{1, k\}$ for some $k \geq 1$ then $a \vdash g$ follows from the conjunction of all $a_i \vdash g_i$.
- If $N = (OR, g, a)$ has a successor $N' = (t', g', a')$ then $a \vdash g$ follows from $a' \vdash g'$.
- If $N = (IF, g, a)$ has successors $N_1 = (t_1, g_1, a_1)$ and $N_2 = (t_2, g_2, a_2)$ then $a_1 \vdash g_1$ implies that $a \vdash g$ follows from $a_2 \vdash g_2$.

These semantic rules have to be observed by each basic prover. Semantically IF-nodes are equivalent to AND-nodes with only 2 successors. They differ operationally, however, in that the first successor of an IF-node has to be proved first in order to produce a proof in natural style.

Definition 5 *An annotation $f : Nodes(T) \to PV$ is called* valid *if f is admissible and for all terminal nodes $N = (t, g, a)$ of T with $t \neq PS$, if $f(N) = proved$ then $a \vdash g$.*

We can now prove the following theorem:

Theorem 2 *Let T be a tree satisfying the semantic rules and $f : Nodes(T) \rightarrow PV$ be a valid annotation. Let $N = (t, a, g)$ be a node in T. The correct annotation f' computed from f has the property that if $f'(N) = proved$ then $a \vdash g$.*

Proof The proof is by induction on the structure of deduction trees. If T consists in just one terminal node then the theorem obviously holds by the definition of valid functions.
For the induction step, let T be a tree with root $N = (t, a, g)$ and let $N_i = (t_i, a_i, g_i)$ with $i \in \overline{1,k}$ be the successors of N. We assume the theorem holds for the deduction trees originating at each of the nodes N_i and we show that it holds for N. We distinguish 3 cases depending on whether the type of N is AND, OR or IF.
Case 1: $t = AND$. If $f'(N) = proved$ then $\Pi(AND, \{f'(N_1), \ldots, f'(N_k)\}) = proved$, hence for all $i \in \overline{1,k}$, $f'(N_i) = proved$. But by the induction hypothesis this implies $a_i \vdash g_i$ for each i. Since T satisfies the semantic rules we get $a \vdash g$.
Case 1: $t = AND$. If $f'(N) = proved$ then $\Pi(OR, \{f'(N_1), \ldots, f'(N_k)\}) = proved$, hence for some $i \in \overline{1,k}$, $f'(N_i) = proved$. By the induction hypothesis this implies $a_i \vdash g_i$ and from the semantic rules we get $a \vdash g$.
The case $t = IF$ is similar to the case $t = AND$.

3.4 Basic Provers in *Theorema*

Basic provers are *Mathematica* programs taking as argument a proof situation and returning either a deduction tree with a valid annotation or *Null* if the special prover cannot handle the given proof situation. Taking advantage of the pattern matching features of *Mathematica* in the implementation of basic provers it is easy to limit the applicability of a basic prover to schemata of proof situations.

For the rest of this chapter we supplement proof situations as defined in Section 3.1 with an additional component, the state, which is an assignment of terms to variables. States are used for transmitting heuristic information between basic provers and between the search mechanism and the basic provers as will be shown later on in this chapter.

Constraints on the format of proof situations passed as argument can be defined as *Mathematica* patterns with conditions.

Following the *Mathematica* syntax, a special prover is a set of one or more definitions of the form:

```
ProverName[ Input ] /; Preconditions := Output;
```

where

`ProverName` is the name of the prover

`Input` is a schemata of *Theorema* proof situations, defined by a pattern in the metalanguage.

`Preconditions` is a set of conditions (in the metalanguage) which must be fulfilled by the proof situation in order for the respective prover to be applicable.

`Output` Is a *Mathematica* expression of arbitrary complexity evaluating to a deduction tree with a valid annotation or *Null* on any possible input to which the prover is applicable.

Note that the powerful pattern-matching mechanism of *Mathematica* allows us to write basic prover rules applying to proof situations (g, a, s) such that e.g.

- $(\exists a) f \in a, C[f]$, or:
- $(\forall a) f \in a, C[f]$

where C is a predicate of the metalanguage. The easy and elegant formulation of such constraints is mainly due to the possibility to do pattern-matching on sequences.

Summarizing, basic provers perform a sequence of deduction steps of type OR, AND and IF on a proof-situation and return the corresponding (valid) deduction tree.

In practice the proof value of a node is stored as a component of the node, hence the basic provers do not return a separate annotation but only the deduction tree itself. For the purpose of the theoretical discussion in the previous section it was more natural however to model proof values through annotation functions.

The definition of a special prover given above comes close to the well known notion of method ([Bundy, 1988]) but is different in as far as it contains no postconditions. Accordingly, we do not do proof planning in the sense that we have separate processes of proof plan construction and proof plan execution.

3.5 The Basic Proof Search Algorithm

The proof search engine combines various valid deduction trees obtained by calling basic provers, into the global deduction tree. (In practice these deduction trees have proof value *pending* for all non-terminal nodes. During proof search, the proof values of the nodes in the deduction tree are computed via Algorithm 1. The proof is completed (i.e. the initial statement is proved or the system reports failure) when the proof value of the root node becomes different from *pending*.

In the following subsections of this section we present the sub-procedures used by the search mechanism and conclude with the implementation of the proof search mechanism itself, in Section 3.5.3.

3.5.1 Computing Proof Values

One operation occurring during proof search is the operation of computing a correct annotation starting with a given valid one.

Lemma 2 assures us that such an annotation exists and is unique. We compute it by Algorithm 1, which is implemented in the function `PropagateProofValue`.

In the use of `PropagateProofValue` our system differs from most other theorem proving systems which employ some form of planning. In the theorem provers we have seen, the proof value is inherently part of the problem state description. From the problem state it can be inferred whether the respective subproof is finished or whether some non-empty set of subgoals still remain to be proved. Such provers hence employ general AI planning techniques applying to the general case of searching for a path in a graph of state descriptions (and state modifications). In the particular case of searching for a proof of a formula, states are sets of subgoals and assumptions and state transformations are deduction steps but the general algorithms stay the same (adapted search heuristics have to be used, of course).

We believe that by our treatment of proof values more informed decisions can be made during proof search and the resulting proofs can be brought closer to human-style proving. Ultimately a large set of experiments on concrete proofs will validate or reject this conjecture.

3.5.2 Testing Provers' Applicability

Another major module in the proof search procedure is the one testing provers' applicability. The function `IsApplicable` tests whether a basic prover (i.e. a *Mathematica* function satisfying the specification given in Section 3.4) is applicable to a given proof situation. Note that given the *Mathematica* functions facilitating access to definitions stored in the current session `DownValues`, `OwnValues`, `UpValues`, the applicability decision can be made by extracting the applicability constraints (in form of patterns) from the function definition. Applicability constraints need not be declared outside the prover definition in order for them to be available to the proof search engine mechanism.

3.5.3 Expanding the Global Deduction Tree

Given the components presented in Sections 3.5.1 and 3.5.2, the search for a proof consists in successively selecting an open proof situation in the deduction tree, selecting an applicable basic prover to call, calling it on the given proof situation and plugging the resulting sub-tree in place of the open proof situation initially considered.

Thus, a simplified search algorithm in *Mathematica*, is as follows:

```
While[GetRootProofValue[$TmaProofObject] === "pending",
    currentPos = AdvanceCurrentSearchPos[currentPos];
    If[currentPos === Null,
       Return[$TmaProofObject]];
    currentProofSit = GetProofSituationAtPos[$TmaProofObject,
```

```
(6)        currentPos, "InternalPosition"];
(7)     applicableProvers = SelectProvers[currentProofSit];
(8)     InsertSubProof[$TmaProofObject, applicableProvers];
(9)    PropagateProofValue[$TmaProofObject];
(10) ];
```

In line (2) the position of an open subgoal to be treated is selected from the global deduction tree (`$TmaProofObject`) by calling the function `AdvanceCurrentSearchPos`. The function `AdvanceProofSituation` returns the next (w.r.t a given order) position of an open subgoal in a deduction tree, after the one passed as argument. Some predefined order functions are available (depth-first and breadth-first) but the user can also implement her own order — see also Section 3.6. A `Null` position signifies that no more open proof situations are left in the deduction tree. The list of provers applicable to the current proof situation is constructed (in `applicableProvers` at line (7)).

The function `SelectProvers` tests provers for applicability (see Section 3.5.2) and takes into account user-defined constraints (Section 3.7.1).

The function `InsertSubProof` calls one or more of the basic provers in the list `applicableProvers` and plugs the result into the proof object. More specifically, `InsertSubProof` works along the following lines:

- If no applicable prover is found then the proof value of the current node is set to "failed".
- If there is more than one applicable prover, then an OR node is formed, whose successor nodes each call one of the provers in `applicableProvers` on the current proof situation.
- If exactly one applicable prover is found, then it is applied to the current proof situation and the resulting subproof, if not `Null`, is plugged into the deduction tree at the current position. A `Null` returned subproof is an indication that the respective prover failed on the given proof situation. In this case the proof value of the current node in the deduction tree is set to "failed".

A maximum search depth can be set in the global proof context (as will be shown in Section 3.6) and thus backtracking to the previous OR node occurs when this maximal search depth is attained.

3.6 Local and Global Proof Context

During proof construction it often happens that the same basic prover is called on several proof situations consecutively and hence it can taylor its heuristic depending on knowledge about previous operations performed in the course of the proof. Also, facts gathered during proof situation analysis by one prover can influence the heuristic of another prover participating in the proof. For these reasons we provide a *local* and a *global proof context*, which can be used for communicating heuristic data between provers and for storing proof search parameters. The proof context allows the user to store and modify an object (*Mathematica* expression) associated with a key and to retrieve a value associated with a key.

From the point of view of lifetime of a proof context variable, the following modes of storing an object are provided:

Persistent: When an object is stored as persistent its value stays the same for the rest of the current proof search process. Other provers can modify the value of the object.

Dynamic: An object defined as dynamic while proving a proof situation at node N of the global deduction tree retains its value only during construction of the subtree with root node N.

Persistent proof context variables are implemented via the global proof context. Dynamic variables are implemented through the local state stored in proof situations.

3.6.1 The Global Proof Context

The global proof context can be modified through the following functions:

AddProofContextElem[*key*, *object*] Associates the object *object* with the key *key*.

GetProofContextElem[*key*] Retrieves the object associated with *key*.

Some keys in the global proof context data structure have a built-in meaning associated as listed below.

The key **SearchDepth** holds a *Mathematica* integer, giving the maximum depth of the global deduction tree holding the current proof. If a branch of the deduction tree increases over the value of the parameter then the respective branch is closed with a terminal node with proof value *failed*.

The key **ProofSitOrder** holds a *Mathematica* predicate which takes as argument 2 proof situations and returns $True$ if the first one is greater than the second one and $False$ otherwise. This predicate is used for specifying the order in which proof situations are expanded during proof search.

3.6.2 The Local Proof Context

The local proof context is implemented by means of the state component of a proof situation. Basic provers which generate subtrees of the global deduction tree should transmit the state they receive as argument to all nodes in the return deduction tree and potentially modify the state.

The local proof context contains the predefined key **ProversHistory**, managed by the proof search mechanism. The associated value is the list of provers which were called in each node of the path ending with the current node. Basic provers can use this value in order to decide which heuristics to apply.

3.7 The Proof Strategy Language

3.7.1 Proof Strategy Clauses

In Section 3.4 we saw that it is possible to attach to a special prover a set of *constraints* inherent to its logic and hence fixed for all proofs, in terms of a schemata of proof situations to which the prover applies.

In addition to these we allow the specification of constraints which can be defined for individual (sets of) proofs. We call these constraints *proof strategy clauses*, or when no confusion is possible, simply *clauses*.

A proof strategy clause is a pair consisting in a condition (a *Mathematica* function taking as argument a proof situation and returning $True$ or $False$) and a prover specification.

Our intention is to provide, through the proof strategy language, a means

PSC	$:=$	•**nonExclusive**[*condition*, *proverSpec*] | •**exclusive**[*condition*, *proverSpec*]
$proverSpec$	$:=$	*proverName* | •**then**[*proverSpec*, *proverSpec*] | •**repeat**[*proverSpec*]
$proverName$	$:=$	the name of a basic prover, complying to the specification in Section 3.4
$condition$	$:=$	the *Mathematica* constant `True` or the name of a *Mathematica* predicate taking as argument a proof situation and returning either $True$ or $False$.

Figure 3.1: Syntax of Proof Stratgey Clauses

of programming the interaction of the provers while leaving unchanged the implementation of each of the basic provers being combined.

Also a typical scenario in the usage of *Theorema* would be the user giving a hint (in a language as close as possible to the one a mathematician uses) to the system for proving the given conjecture. Such a user hint need not be an exact specification as to what *Theorema* provers should be called and in what order for proving the conjecture, but could rather be translated into a set of proof strategy clauses as we will show in the remainder of this section. This set of clauses does not necessarily determine uniquely the path in the deduction tree which leads to a solution. Rather several paths would have to be examined in order to prove the conjecture.

The syntax of a proof strategy clause PSC is given in Figure 3.1.

The function `AddClauses` is provided for adding a set of proof strategy clauses (complying to the syntax above) to the clauses to be used in proof search from the point of definition on. The function `ClearClauses` clears all defined clauses.

We describe the semantics of the proof strategy language first by showing how the defined clauses influence the search mechanism in Section 3.5.3. Proof strategy clauses come into play in the process of selecting the provers applicable to a certain proof situation (function `SelectProvers`).

Given a proof situation P, we say that P satisfies a proof strategy clause (c, p) if and only if $c[P]$ returns $True$ and the prover specified by p is applicable to P.

We will define later how the definition of applicability of a basic prover is extended to define applicability of a prover specification.

Given a proof situation P, the function `SelectProvers` selects a list of provers applicable to P along the following steps:

1. Satisfiability of conditions in clauses tagged with •`exclusive` is tested on P. As soon as one satisfiable clause is found, the function `SelectProvers` returns its prover specification.
2. Satisfiability of each clause tagged as •`nonExclusive` is tested on P. For each clause which is found to be satisfiable, the corresponding prover specification is put into a list.
3. The list of all provers found at point 2 is returned.

3.7.2 Tacticals

The tacticals •`then` and •`repeat` were introduced in an attempt to keep the implementation of the basic provers in *Theorema* completely free of prover interaction code. It should thus be possible to specify all situations of prover interaction by means of the proof strategy language. Of course, given the local proof context (state) the tacticals •`then` and •`repeat` could also be programmed in each basic prover by setting and testing variables in the local proof context. This however, would imply, that prover code is being duplicated in several provers' interaction situations.

To avoid this we propose the usage of the tacticals •`then` and •`repeat` whose meaning are described below:

Given the provers A and B, $C = \bullet then[A, B]$ is a prover such that:

- For all proof situations P, $IsApplicable[C, P] = IsApplicable[A, P]$.
- For all proof situations P, C applied to P returns the deduction tree obtained by applying A to P and then applying B to all open proof situations in the resulting deduction tree.

Given a prover A, $C = \bullet repeat[A]$ is a prover such that:

- For all proof situations P, $IsApplicable[C, P] = IsApplicable[A, P]$.

- For all proof situations P, C applied to P returns the deduction tree obtained by applying A to P and then repeatedly applying A to all open proof situations in the resulting deduction tree, until each call returns `Null`.

At the time of this writing the tacticals above are not yet implemented in the provers' interaction mechanism. However, in the proof examples given in the remainder of this thesis we illustrate their use in reducing the proof search space.

3.8 Using the Mechanism for Provers' Interaction

The basic provers available in *Theorema* to which we would like to apply the mechanism described above are:

- Predicate Logic Prover (in Natural Deduction Style) [Buchberger, Jebelean, 1997],
- Induction Prover for Tuples [Buchberger, Vasaru, 1997] and Natural Numbers [Buchberger, 1997],
- Simplification Prover [Buchberger, Marin, 1997],
- Functor Prover [Tomuta, 1997].

The predicate logic prover can prove theorems in higher order logic and it is based on a set of rules which imitate the human style of proving (see [Buchberger, Lichtenberger, 1981]).

The simplification prover attempts to prove equalities by reducing both terms of an equality to the same canonical form with respect to a set of equalities passed as argument. The prover is able to handle associative and commutative operators.

The functor prover is a collection of separate procedures for applying definitions of domain operators and definitions of domain properties (as will be shown in Chapter 4).

When doing induction over lists, the simplification prover is first called in an attempt to prove the given conjecture and then induction is done recursively over each universally quantified variable in the goal until the theorem is proved, or until all variables are tried out. The prover for equalities over the natural numbers works in a similar manner.

Several other provers, each a combination of the existing basic provers listed above are planned (or already available), according to the basic *Theorema* philosophy [Buchberger, 1996c] of providing special provers for each domain. We will not deal in the remainder of this chapter with the case of domains built up by functors, which are presented in the following chapter.

However, a special prover will also be provided for the primitive domain of sets. This prover should use (parts of) the predicate logic prover and some set specific inference rules.

In parallel with the work on designing and implementing a functor language [Buchberger, Kriftner, 1997] and with using functor knowledge for simplifying the complexity of proving [Tomuta, 1997], some specialized proof techniques for certain domains of mathematics have been studied: A prover for theorems in analysis is under development and a prover for the domain of arrays of natural numbers (to be used in the verification of parallel programs) is planned. Both of these provers rely on the existing basic provers.

3.9 An Example

To see the interaction mechanism at work, we give below a proof generated by two basic provers: the induction prover and the simplification prover.

Consider the construction of natural numbers based on the Peano axioms, with initial element 0 and the successor of an element n denoted by n^+. The theorem below proves that addition is commutative. The definition of addition as well as some properties needed in the proof are given in the statement of the theorem at the beginning of the proof.

The text below was converted to LaTeX from a *Mathematica* notebook by using the data export facilities provided by *Mathematica* . The exported text does not show the cell structure of the notebook. We have added line numbers for easier reference.

---

(1) Prove: (+c) $\forall_{m,n}(\boldsymbol{m}+\boldsymbol{n}=\boldsymbol{n}+\boldsymbol{m})$,
(2) under the assumptions:
(3) (+0) $\forall_{m}(\boldsymbol{m}+0=\boldsymbol{m})$,
(4) (0+) $\forall_{m}(0+\boldsymbol{m}=\boldsymbol{m})$,
(5) (++r) $\forall_{m,n}(\boldsymbol{m}+\boldsymbol{n}^{+}=(\boldsymbol{m}+\boldsymbol{n})^{+})$,
(6) (++l) $\forall_{m,n}(\boldsymbol{m}^{+}+\boldsymbol{n}=(\boldsymbol{m}+\boldsymbol{n})^{+})$.
(7) There are 2 alternatives for proceeeding with the proof.
(8) Alternative 1: by simplification
(9) Sorry, proving (+c) fails.
(10) Alternative 2: by induction.
(11) We prove (+c) by induction on $\boldsymbol{m}$.
(12) Induction Base:
(13) (+c.IB) $\forall_{n}(0+\boldsymbol{n}=\boldsymbol{n}+0)$.
(14) There are 2 alternatives for proceeeding with the proof.
(15) Alternative 1: by simplification
(16) We take in (+c.IB) all variables arbitrary but fixed and prove:
(17) (+c.IB.F) $0+\boldsymbol{n}_1=\boldsymbol{n}_1+0$.
(18) A proof by simplification of (+c.IB.F) works.
(19) Simplification of the lhs term:
(20) $0+\boldsymbol{n}_1=$ by (0+)
(21) $\boldsymbol{n}_1\rfloor$
(22) Simplification of the rhs term:
(23) $\boldsymbol{n}_1+0=$ by (+0)
(24) $\boldsymbol{n}_1\rfloor$
(25) Alternative 2: by induction
(26) Pending proof of (+c.IB).
(27) Induction Hypothesis:
(28) (+c.IH) $\forall_{n}(\boldsymbol{m}_1+\boldsymbol{n}=\boldsymbol{n}+\boldsymbol{m}_1)$
(29) Induction Step:
(30) (+c.IS) $\forall_{n}((\boldsymbol{m}_1)^{+}+\boldsymbol{n}=\boldsymbol{n}+(\boldsymbol{m}_1)^{+})$.
(31) There are 2 alternatives for proceeeding with the proof.
(32) Alternative 1: by simplification
(33) We take in (+c.IS) all variables arbitrary but fixed and prove:
(34) (+c.IS.F) $(\boldsymbol{m}_1)^{+}+\boldsymbol{n}_2=\boldsymbol{n}_2+(\boldsymbol{m}_1)^{+}$.
(35) A proof by simplification of (+c.IS.F) works.
(36) Simplification of the lhs term:
(37) $(\boldsymbol{m}_1)^{+}+\boldsymbol{n}_2=$ by (++l)

(38) $\quad (\boldsymbol{m}_1 + \boldsymbol{n}_2)^+ \rfloor$
(39) Simplification of the rhs term:
(40) $\quad \boldsymbol{n}_2 + (\boldsymbol{m}_1)^+ =$ by (++r)
(41) $\quad (\boldsymbol{n}_2 + \boldsymbol{m}_1)^+ =$ by (+c.IH)
(42) $\quad (\boldsymbol{m}_1 + \boldsymbol{n}_2)^+ \rfloor$
(43) Alternative 2: by induction
(44) Pending proof of (+c.IS).

The proof strategy clauses used for generating the proof shown above were set with the following command:

```
AddClauses[  •nonExclusive[True, Simplifier],
             •nonExclusive[True, NIP]];
```

The two basic provers `Simplifier` and `NIP` perform proofs by simplification and induction on natural numbers respectively.

Both provers are applicable to the initial proof situation, a fact which is stated in line 7 of the proof listing. Lines 8 and 9 contain the failed attempt by the simplifier to transform the proof situation. In lines 10 to 13 induction is applied and the initial proof situation is split into 2 new proof situations: the base case, shown in line 13 and the induction step in lines 27 to 30. For the proof of the base case, again both provers are applicable. The proof by simplification succeeds (lines 18 to 24). Lines 25 and 26 correspond to the open base case proof situation to be proved by induction. This proof is not done anymore because the proof by simplification succeeded and the parent OR node of this proof situation gets proof value *proved*, before the alternative proof is started. The same situation occurs in the proof of the induction step. The proof by simplification succeeds (lines 32 to 42) and the proof by induction which is not anymore relevant is shown only as an open proof situation cell (lines 42, 43).

As a parenthesis, the appearance of the proof listed above as a *Mathematica* notebook is greatly enhanced by the possibility to group text in cells. The cell structure of the proof corresponds to the structure of the deduction tree. For instance, an OR node or and AND node with 3 successor nodes in the deduction tree are displayed as a cell with 3 sub-cells, each corresponding to one of the 3 successor nodes. Linear parts of the proofs (e.g. AND nodes with only one successor) appear as successive cells with no nesting.

In the sample proof above several OR nodes occur which contain 2 successor nodes out of which one has proof value *proved* and the other one is either *pending* or *failed*. When there are to many such nodes the proof presentation becomes cluttered. The problem can be addressed in two ways:

- by doing post-processing of the deduction tree such that failed or pending branches of the proof are removed from the final deduction tree and OR nodes with one proved successor node are replaced with the respective successor.
- by modifying the set of proof strategy clauses to eliminate some of the OR nodes at the time when the proof is built (thus reducing the search space).

Concerning the second approach, we can reduce the search space by employing the tacticals `•then` and `•repeat`. The definition of the set of clauses will then be as follows:

```
AddClauses[•nonExclusive[True, •then[Simplifier, NIP]]]
```

With this set of clauses the simplifier is called on the initial proof situation and leaves it unchanged because it fails by returning `Null`. Next, the induction prover for natural numbers is called and produces an AND node containing the induction base and the induction step. On each of these proof situations, the prover `•then[Simplifier, NIP]` is called and finalizes the proof. The resulting deduction tree contains no OR nodes.

3.10 Integrating Interactivity

Currently proofs in *Theorema* are generated completely automatically. The aim is however, to offer the user mixed interactive-automated style proving facilities. Ideally, the user should be asked for hints in the difficult or ingenious parts of the proof. Knowing when to ask for hints is, of course a difficult problem and it depends strongly on the mathematical domain we work in and on the type of proofs we tackle. We do not discuss this problem here but merely show how interactive proving can be integrated into the current architecture for provers cooperation presented in this chapter and what support for interaction is already implemented.

The algorithm in Section 3.5.3 consists in a loop which selects the next proof situation to be treated and the prover(s) to be called for solving it. A step-by-step version of this function exists, consisting in the following procedures:

`StartProof` initializes all necessary data structures and creates a global deduction tree containing just the initial proof situation.

`AdvanceCurrentSearchPos` advances the proof situation to be treated to the next one in a list of open proof situations ordered by the current order on proof situations

`OneProofStep` performs one step in the proof, by selecting a the current proof situation (as set by `AdvanceCurrentSearchPos`), calling a prover on it and plugging the result into the global deduction tree. If more than one prover is applicable then the function simply inserts an OR node into the global proof situation as shown in Section 3.5.3.

After each execution of the three procedures above the global deduction tree holds the current partial proof which can be displayed to the user by calling the proof display functions.

The immediate use of these procedures is for allowing a step-by step execution of the proof, where, after each step, the user sees the new stage of the proof. To allow for more control, the function `AdvanceProofSearch` can be replaced with an interactive version which asks the user to select the next proof situation to tackle if more than one such proof situations exist.

Interactivity is also possible at the level of basic provers. Recall that the main requirement for basic provers is that they return a deduction tree with a valid annotation. The implementation of the prover can be such that the prover asks the user which of a set of applicable deduction steps to apply to the current proof situation, then protocols the result in a valid deduction tree and returns it.

For the proof search mechanism such behavior is transparent. Such interactive provers can be integrated into the interaction mechanism by specifying proof strategy clauses for them as shown in Section 3.7.1. The proof will then be performed automatically until the point when an interactive prover becomes applicable and is called. The challenging problem will then be to design the proof strategy clauses for calling interactive provers such that the right amount of interaction is achieved. Taking this idea even further the basic interactive provers can have as effect a change in the current strategy of the prover by modifying the current set of clauses based on input from the user.

Summarizing, we think that the provers interaction mechanism offers basic ingredients for making *Theorema* a mixed automated-interactive theorem prover.

Chapter 4

A Functor Language

For the purpose of experimenting with proofs in domains built up by functors, we have implemented a toy functor language allowing the user to define *functors*, *domains* and *domain properties*. A much more powerful language is being developed in parallel in the *Theorema* project (see [Buchberger, Kriftner, 1997]). Our motivation however was to identify the problems that appear when proving statements involving domains. We developed some proof procedures handling the constructs of the functor language and we applied them to proving knowledge transport properties as will be shown in the next chapter. Given that the *Theorema* system aims at combining proving and computing a functor definition has both a computational and a logical aspect. In the work presented below we are not concerned with the computational aspect but only with the logical one. When a functor is defined, the formulae defining its operators are transformed into *methods* and stored in an internal representation of the functor. This facilitates manipulating definitions during the proof search process. Although the language we present in this section is at times cumbersome to use and will not be included in its current form in *Theorema* the constructs used for proving can probably be adopted in a very similar form in a more sophisticated language. The separation of the computational and proving context in *Theorema* as well as the independence of the internal representation of a functor used by our prover from the syntax of the functor definition allows the constructs we present to be supplemented with computational content features.

Also, our experiments highlighted some problems in proving with functors which influenced the design of the deduction tree data-structure presented in the previous chapter.

At the time of this writing the deduction tree data structure is not yet integrated with the functor prover presented in Section 4.7. We explain how the functor prover currently works and show what benefits could be derived from integrating it with the proof-search mechanism in Chapter 3.

4.1 Defining Functors

A functor is a collection of formulae describing the way in which operators in a resulting domain depend on operators in some input domains.

The syntax for functor definitions is as follows:

$$
\begin{array}{lll}
FunctorName[D_1, \ldots, D_n] \equiv & \textbf{Functor}[& \{D,\ operators\}, \\
 & & F_1, \\
 & & F_2, \\
 & & \vdots \\
 & & F_n, \\
 &] &
\end{array}
$$

where

- $FunctorName$ is the name of the functor we are defining;
- D_1, ..., D_n are the formal parameters of the functor definitions, i.e. the list of input domains;
- D is the output domain;
- $operators$ is a *Mathematica* sequence of operators defined in the functor $FunctorName$ (the signature of the defined domain D);
- F_1, ..., F_n are labeled formulae defining the operators of the resulting domain D.

Note that a functor definition only encapsulates definitions of domain operators and does not contain any axioms characterizing the resulting domain. Such axioms and additional properties can be stated by means of the domain property construct presented in Section 4.3.

The carrier set of the resulting domain D (see also [Buchberger *et al.*, 1997b]) is defined by means of the distinguished unary operator $\underset{D}{\epsilon}$.

An operator defined on the left hand side of a functor definition can be a predicate, a constant or a function.

The recognizer `IsFunctor` returns $True$ if its argument is a functor name and $False$ otherwise.

4.1.1 Defining Operators

The functor language parser supports predicate definitions of the following forms:

$$\forall_{\{x_1,\ldots,x_n\}} c(x_1,\ldots,x_n) \Rightarrow (\underset{D}{\text{pred}}[x_1,\ldots,x_n] \Leftrightarrow rhs(x_1,\ldots,x_n)) \tag{4.1}$$

or

$$\forall_{\{x_1,\ldots,x_n\}} \underset{D}{\text{pred}}[x_1,\ldots,x_n] \Leftrightarrow rhs(x_1,\ldots,x_n) \tag{4.2}$$

Where *pred* is the name of the operator being defined, x_1, ..., x_n are symbols and by $f(x_1,\ldots,x_k)$ we denote a *Theorema* formula f (possibly containing quantifiers) with free variables among $x_1,\ldots,x_k$. Since we do not deal with the computational content of domains in this thesis, no restrictions are posed on the formulae $rhs(x)$ and respectively $rhs(x_1,\ldots,x_n)$ which may, in general, be undecidable. In particular, if *pred* is ϵ then the definition formula is treated in a special way as will be shown later on in this chapter.

Similarly, the definition of a functional operator f of the domain D, may be of the form:

$$\forall_{\{x_1,\ldots,x_n\}} \underset{D}{\text{f}}[x_1,\ldots,x_n] \equiv rhs(x_1,\ldots,x_n) \tag{4.3}$$

where $rhs(x_1,\ldots,x_n)$ is a term with variables among $x_1,\ldots,x_n$.

In particular, a function with no arguments is a constant, and the corresponding syntax for defining a constant *const* in D is:

$$\underset{D}{\text{const}} \equiv rhs \tag{4.4}$$

with rhs a ground term.

For example the definition of the functor defining the cartesian product of two domains is as follows:

Example 1

$$\begin{aligned}&\mathrm{CartesianProduct}[A,\ B]\ \equiv\\&\mathrm{Functor}\,[\langle CP,\ \epsilon\rangle,\\&\langle\text{"}\epsilon\text{"}, \forall_{\langle x\rangle}\ \underset{CP}{\epsilon}\ [x]\ \Leftrightarrow\ \exists_{\langle a,b\rangle}\ \underset{A}{\epsilon}\ [a]\ \wedge\ \underset{B}{\epsilon}\ [b]\ \wedge(x\ \equiv\ \langle a,\ b\rangle)\rangle;\\&\ ]\end{aligned}$$

We can also define 3 operations, $\oplus$ and $\ominus$ component-wise and a constant o as follows:

Example 2

$$\begin{aligned}&\mathrm{CartesianProduct}[A,\ B]\ \equiv\\&\mathrm{Functor}\,[\langle CP,\ \epsilon,\oplus,\ o,\ \ominus,\ \rangle,\\&\langle\text{"}\epsilon\text{"}, \forall_{\langle x\rangle}\ \underset{CP}{\epsilon}\ [x]\ \Leftrightarrow\ \exists_{\langle a,b\rangle}\ \underset{A}{\epsilon}\ [a]\ \wedge\ \underset{B}{\epsilon}\ [b]\ \wedge(x\ \equiv\ \langle a,\ b\rangle)\rangle;\\[2ex]&\langle\text{"}\oplus\text{"}, \forall_{\langle a1,\ b1,\ a2,\ b2\rangle}\langle a1, b1\rangle\ \underset{CP}{\oplus}\ \langle a2,\ b2\rangle\equiv\ \langle a1\ \underset{A}{\oplus}\ a2,\ b1\ \underset{B}{\oplus}\ b2\rangle\rangle;\\[2ex]&\langle\text{"}o\text{"},\ \underset{CP}{\mathrm{o}}\ \equiv\langle\ \underset{A}{\mathrm{o}}\ ,\ \underset{B}{\mathrm{o}}\ \rangle\rangle;\\[2ex]&\langle\text{"}\ominus\text{"}, \forall_{\langle a,\ b\rangle}\ \underset{CP}{\ominus}\ [\langle a,\ b\rangle]\ \equiv\ \langle\ \underset{A}{\ominus}\ [a],\ \ \underset{B}{\ominus}\ [b]\rangle\rangle;\\[2ex]&\ ]\end{aligned}$$

The example above shows the possibility of using pattern matching when defining domain operators.

Let us assume, for instance, that a domain C was defined as the `CartesianProduct` of two domains A and B (the corresponding language construct will be introduced in Section 4.2). Then, given a term $a \underset{C}{\otimes} b$, the definition of C is applicable to the term only if both a and b match a pattern of the form $\{\_,\ \_\}$.

In an operator definition of one of the forms 4.1, 4.2, 4.3, 4.4 if the "formal parameters", x, x_1, ..., x_n have no specified type (i.e. no restriction of the form $\underset{Domain}{\epsilon}[x]$ is placed on them, then they are assumed to be of the type of the output domain.

Furthermore, an operator definition of arity k can be of the form:

$$\begin{gathered}\forall_{\{x_1,\ldots,x_n\}} c(x_1,\ldots,x_n) \Rightarrow \\ pred[t_1(x_1,\ldots,x_n),\ldots,t_k(x_1,\ldots,x_n)] \Leftrightarrow rhs(x_1,\ldots,x_n)\end{gathered} \tag{4.5}$$

or

$$\begin{aligned}\forall_{\{x_1,\ldots x_n\}} & fun[t_1(x_1,\ldots,x_n),\ldots,t_k(x_1,\ldots,x_n)] \\ & \equiv rhs(x_1,\ldots,x_n)\end{aligned} \tag{4.6}$$

with the condition $c(x_1,\ldots x_n)$ possibly missing in 4.5. The free variables in the terms $t_1(x_1,\ldots,x_n)$, ..., $t_k(x_1,\ldots,x_n)$ are among $x_1,\ldots,x_n$.

For instance, in the definition of $\otimes$ in the example above, t_1 and t_2 are both equal to the tuple operator $\langle,\rangle$.

When a functor definition is evaluated, the body of the functor is parsed and a corresponding internal representation is generated, which will be described in more detail in Section 4.7. Afterwards, calling `IsFunctor` on the symbol newly defined to be a functor will return $True$.

4.2 Defining Domains

A domain can be defined by means of the function $DefineDomain$. Assuming that the functor $FunctorName$ of arity n has been defined, we can enter the command:

DefineDomain[D, $FunctorName$[D_1, ..., D_n]]

to define the domain D in terms of the existing domains D_1, ..., D_n. When such a definition is entered, any previous functor definition associated with

D is deleted. The "type" of D as well as the types of each of the symbols $D_1, \ldots, D_n$ are set to *domain*.

The function *IsDomain* checks the type of a symbol and returns $True$ if its argument is a domain and $False$ otherwise.

For example, having entered the definition of the functor `CartesianProduct`, we can use it as follows:

$$\text{DefineDomain}[\boldsymbol{C},\ \text{CartesianProduct}[\boldsymbol{A},\ \boldsymbol{B}]];$$

$$\text{IsDomain}[\boldsymbol{A}]$$

$$True$$

$$\text{IsDomain}[\boldsymbol{B}]$$

$$True$$

$$\text{IsDomain}[\boldsymbol{C}]$$

$$True$$

4.3 Domain Properties

Domain properties can be defined by means of a language construct which allows the user to associate a name to a list of *Theorema* formulae involving operators of some domain. The construct allows parameterization with respect to a domain name and some domain operators. Our aim when providing this construct was to allow the user to formulate and store algebraic properties of domains like "G is a group with respect to the operations $+$ and $\times$".

Once the user has defined some domain properties she can state that a domain has the respective property with respect to some of its operators.

In the remainder of this section we show how domain properties can be defined and how they can be associated to existing domains.

4.3.1 Defining Properties

A *domain property* definition has the form:

$$PropertyName[dom, operators] \equiv \{F_1, F_2, \ldots, F_n\}$$

where

- $PropertyName$ is the name of the property being defined;
- dom is a formal argument in the domain definition, representing the domain to which we associate the property $PropertyName$;
- $operators$ is a sequence of formal parameters which, at the time of applying the domain property to a concrete domain will be instantiated with operators of the respective domain;
- $F_1, \ldots, F_n$ are labeled *Theorema* formulae.

For instance we can define the property which states that a domain G is a group with respect to the operations of addition, neutral element and taking the inverse of an element as follows:

$$\text{Group}[G, \oplus, o, \ominus] \equiv$$

$$\langle\langle\text{"}\oplus.closure\text{"}, \forall_{\langle x, y\rangle}\Big(\Big(\underset{G}{\epsilon}[x] \wedge \underset{G}{\epsilon}[y]\Big) \Rightarrow \underset{G}{\epsilon}[x \oplus y]\Big)\rangle,$$

$$\langle\text{"}\ominus.closure\text{"}, \forall_{\langle x\rangle}\Big(\Big(\underset{G}{\epsilon}[x]\Big) \Rightarrow \underset{G}{\epsilon}[\ominus[x]]\Big)\rangle,$$

$$\langle\text{"}\oplus.assoc\text{"},$$

$$\forall_{\langle x, y, z\rangle}\Big(\underset{G}{\epsilon}[x] \wedge \underset{G}{\epsilon}[y] \wedge \underset{G}{\epsilon}[z]\Big) \Rightarrow (x \oplus y) \oplus z \equiv x \oplus (y \oplus z)\rangle,$$

$$\langle\text{"}\oplus.o.left\text{"}, \forall_{\langle x\rangle} \underset{G}{\epsilon}[x] \Rightarrow (o \oplus x \equiv x)\rangle,$$

$$\langle\text{"}\oplus.\ominus.left\text{"}, \forall_{\langle x\rangle} \underset{G}{\epsilon}[x] \Rightarrow (x \oplus \ominus[x] \equiv o)\rangle$$

$$\rangle;$$

Clearly such a construct is to weak for a "real" functor language. For instance it is often desirable to be able to formulate a property involving several domains, just as for the functor construct it would be desirable to be able to parameterize a functor also by operators or elements of the carrier set not just by input domains. The property of a domain being a vector space with respect to some domain of scalars or the functor supplementing an existing domain with new operators are examples of such situations.

This, however, introduces the need of some kind of "type" declaration allowing us to distinguish between the arguments to the functor (or property) definition which are domains, operators and elements of the carrier set. In the simple language constructs we introduced we eluded this problem by inferring the type of a formal parameter from its position in the argument list when parsing property definitions.

Only a syntactic check is performed when a property definition is parsed. Namely, it is checked that the domain operators which appear as formal arguments from the second position on, are of the form $\underset{dom}{\mathrm{symb}}$ where *symb* is a symbol and *dom* is the first formal argument. The form of the formulae composing the definition is not restricted in any way.

Defined properties are stored internally in *Theorema* and can be used throughout the current session. A recognizer is available which allows the user to test if an existing symbol is a property name. If a new property definition is entered for an already existing property name than the old definition is replaced by the current one. A property definition can be deleted by means of the function `DeleteProperty`.

Hence, assuming that the definition of the `Group` property was entered in the current session, we could interact with *Mathematica* as follows:

$$\mathrm{IsProperty}[\boldsymbol{Group}]$$

$$True$$

$$\mathrm{DeleteProperty}[\boldsymbol{Group}];$$

$$\mathrm{IsProperty}[\boldsymbol{Group}]$$

$$False$$

4.3.2 Using Property Definitions

To specify that a defined property holds for a given domain, the user can use the function `Assume`, with the following syntax:

$$\mathbf{Assume}[PropertyName[dom, operators]]$$

where

- $PropertyName$ is the name of an existing property;
- dom is the name of an existing domain for which we state that is has property $PropertyName$
- $operators$ is a sequence of operators of dom, i.e. of expressions of the form $\underset{dom}{\text{symb}}$ where $symb$ is a symbol denoting an operator in dom.

The function `Assume` checks whether its first actual argument is a domain, by using the function `IsDomain`. The following formal arguments, i.e. the operators of the respective domain need not appear in the signature of the domain listed in the functor definition. The signature only lists the symbols for which a definition is actually given within the functor body. Through the domain property construct it is possible to formulate properties of operators for which a definition is not available. Roughly, this corresponds to the mathematical formulation: " There exists an operator o in the domain D such that o satisfies the property P".

Several properties can be associated to a given domain by means of the function `Assume`. If a property is deleted by using the function `DeleteProperty` then all assumptions involving the respective property and existing domains are deleted.

4.4 Constructs of the Functor Language

Summarizing, the functor language we present in this chapter, supplements the *Theorema* language with the following constructs:

- functor definitions (introduced in Section 4.1)

- domain definitions (introduced in Section 4.2)
- property definitions (introduced in Section 4.3.1)
- assuming properties about domains (introduced in Section 4.3.2)

In addition, the following are atomic formulae of the *Theorema* logical language:

- If $FunctorName$ is a functor and D, D_1, ..., D_n are variables, then

  $$D \equiv FunctorName[D_1, \ldots, D_n]$$

  is a formula;

- If $PropertyName$ is a domain property and D, op_1, ..., op_k are variables, then

  $$PropertyName[D, \underset{D}{\mathrm{op_1}}, \ldots, \underset{D}{\mathrm{op_k}}]$$

  is a formula.

4.5 The Functor Prover

The call to the functor prover is of the form:

$$\mathbf{FunctorProver}[F, \{F_1, \ldots F_n\}]$$

where

- F is the formula to be proven
- F_1, ..., F_n is the list of assumptions, with F_i being *Theorema* formulae with the extensions introduced in the previous section.

We will now describe the inference rules applied by the functor prover. This also gives semantics to the functor language constructs listed in the previous section.

In a version of the system which combines computation with deduction, domain definitions will also associate a computational meaning to the domain's operators, enabling the user to evaluate terms of the form:

$$\underset{Dom}{\mathrm{op}}\ [a_1, \ldots, a_k] \tag{4.7}$$

for a an operator *op* in a domain *Dom*.

In the current stage though, we are only concerned with proving properties of domains, so the domain definitions, via `DefineDomain` and the `Assume` function are provided only as a replacement for functor language formulae in the knowledge base.

In other words, if a domain definition of the form

$$DefineDomain[D, FunctorName[D_1, \ldots, D_n]]$$

was entered in the current session, then proving a formula F for a set of assumptions KB is the same as proving F from

$$KB \bigcup \{D \equiv FunctorName[D_1, \ldots, D_n]\}.$$

Similarly, if a statement of the form

$$Assume[PropertyName[D, \underset{D}{\mathrm{op_1}}, \ldots, \underset{D}{\mathrm{op_n}}]]$$

has been entered in the current session, then proving formula F for a set of assumptions KB is the same as proving F from

$$KB \bigcup \{PropertyName[D, \underset{D}{\mathrm{op_1}}, \ldots, \underset{D}{\mathrm{op_n}}]\}.$$

Hence, the set of assumptions used during a proof are composed of the formulae explicitly given as assumptions in the prover call plus knowledge derived from domain definitions and properties assumed about domains in the current session.

During a proof, the functor prover applies some transformations to the goal and knowledge base, which are of the following 3 types:

- It *replaces domain properties* with a conjunction of formulae with the domain and operator arguments properly instantiated, i.e. it applies domain properties definitions;

- It applies definitions of domain operators;
- It uses *type* information for terms which are known to be in the carrier set of a certain domain (i.e. it applies the definition of the membership operator of a domain);

Below we describe these operations in more detail, thereby giving a meaning to the functor language constructs presented so far.

In the presentation below, for simplicity, we leave out the labels of *Theorema* formulae.

Let g be the current formula to be proven and kb be the current list of assumed *Theorema* formulae (the knowledge base). We will consider kb to be a set of formulae, since by using the sequence pattern matching feature of *Mathematica* , we can directly refer to an element in kb which satisfies a certain condition and need not implement the usual rules for swapping elements or bringing an element on the first position of the list of assumptions.

The transformations applied by the functor prover to the pair (g, kb) are as follows:

Expand property definition

If

- $F \equiv PropertyName[D, \underset{D}{\mathrm{o}_1}, \ldots, \underset{D}{\mathrm{o}_\mathrm{n}}]$ is a formula in $\{g\} \bigcup kb$
- $PropertyName$ is a domain property defined as:

$$PropertyName[X, op_1, \ldots, op_n] \equiv \{F_1, \ldots, F_k\}$$

then replace F with the formula $\sigma(F_1 \wedge \ldots \wedge F_k)$ where $\sigma = \{X \leftarrow D, op_1 \leftarrow \underset{D}{\mathrm{o}_1}, \ldots, op_n \leftarrow \underset{D}{\mathrm{op}_\mathrm{n}}\}$

Apply operator definition

If

- $\{D \equiv FunctorName[D_1, D_2, \ldots, D_n]\} \subset kb$ and $FunctorName$ is the name of a functor defined in the current session,
- some formula F in $\{g\} \bigcup kb$ contains a sub-term $\underset{D}{\mathrm{op}}[t_1, \ldots, t_n]$,

then replace all occurrences of $\underset{D}{\mathrm{op}}\,[t_1,\ldots,t_n]$ in F with the definition of $\underset{D}{\mathrm{op}}$ applied to the terms $t_1,\ldots,t_n$, under assumptions kb.

Use "type" information
If

- $D \equiv FunctorName[D_1, D_2, \ldots, D_n] \in kb$ and $FunctorName$ is the name of a functor defined in the current session,
- some formula F in $\{g\} \bigcup kb$ contains a sub-term t,
- $\underset{D}{\epsilon}\,[t] \in kb$,

then apply type knowledge to F and kb.

We still have to specify what is meant by applying type knowledge and by applying operator definitions and we do this in the following section.

4.6 Methods

The way in which operator definitions are applied can best be explained by means of a concept used in the implementation of the prover, namely the one of *method*. A concept with this name was introduced by Bundy (see [Bundy, 1988]), who attached "specifications" to tactics allowing a proof-planner to build a plan of a proof by working only with specifications and without actually executing the tactics. Methods in the sense of Bundy are tuples composed of preconditions, input, output and postconditions descriptions. These are also the components of the methods in the functor prover. However, in their use the functor prover methods are quite different. We transform logical formulae (definitions) into "rules" (or methods) i.e. into an intermediate format in order to make application to terms easier. This strategy goes in the direction of expert systems and was used in some older systems aiming to imitate human proof techniques, like MUSCADET ([Pastre, 1989]).

Functor definitions are parsed and stored in an internal format which in addition to the logical formulae defining operators also stores one or more methods corresponding to each such definition.

In the remainder of this section we will explain how type knowledge and operator definitions are applied by using the method notation. The 2 operations we describe each apply a method to a formula in presence of a knowledge base and possibly change the knowledge base. Hence we need to explain the following 3 aspects:

1. how methods are generated from operator definitions given in section 4.1.1,
2. what it means for a method to be applicable,
3. how a method is applied to a formula under a set of assumptions kb.

We start with points 2 and 3 and we will treat question number 1 in the following 2 sections for the 2 cases of applying type knowledge and applying operator definitions.

A method is a tuple $(V, Pre, In, Out, Post)$ with components having the following meaning:

V is a list of variables,

Pre is a list of (unlabeled) formulae called preconditions,

In, Out are terms representing the input and output of the method,

$Post$ is a list of unlabeled formulae called postconditions.

A method $(V, Pre, In, Out, Post)$ is applicable to a term t under the assumptions kb if there exists a substitution σ of terms for the variables V such that $\sigma(In)$ is a sub-term of t at some position pos and $\sigma(Pre)$ is included in kb.

The applicability test of a method is implemented by using the *Mathematica* pattern matching mechanism for finding the substitution σ. A consequence is that we always find one substitution and thus one way to apply the method to the term although several such substitutions may exist.

Let t be a term, let $M = (V, Pre, In, Out, Post)$ be a method which is applicable to t under the assumptions kb and let σ be the substitution of terms for the variables V such that the sub-term of t at position pos is equal to $\sigma(In)$. As a result of applying M to t under the assumptions kb, the pair (t, kb) is transformed into (t', kb') such that:

- t' is obtained from $\sigma(t)$ by replacing the sub-term at position *pos* with $\sigma(Out)$.
- $kb' = kb \bigcup \sigma(Post)$

As stated before, because we rely on the pattern-matching mechanism of *Mathematica* for finding positions in a term at which a method is applicable, we are restricted in the choice of the position at which to apply the method. In case several such positions exist, we will, by repeated method application exhaust them, but the order in which the positions are considered is dictated by the pattern matching mechanism. This however was not a hindrance in the experiments we did so far (see Chapter 5).

The generation of methods from operator definitions is rather straightforward. The membership operator is treated differently from usual operators, in that additional methods are generated which enable an element in the carrier set of a domain to be expressed in terms of elements in carrier sets of the underlying domains. We start with the case of methods used for applying operator definitions.

Note that when applying operator definitions the functor prover only works with methods and the operator definitions as input by the user are only stored for displaying purposes. The prover is thus independent of the syntax of functor definitions. If the syntax changes, then only the parser, translating operator definitions into methods will need to be changed.

4.6.1 Applying Operator Definitions

We now describe how methods are generated from operator definitions and in doing this we will consider only the most general forms of the definitions given in Section 4.1.1. For each operator definition at least 2 methods are generated, which correspond to the cases in which the operator whose definition we apply is part of the goal or of the knowledge base. For functional operators and constants, the two methods are identical. For predicates however, they are different, corresponding to the case of applying the definition of a predicate we want to prove and, respectively, applying the definition of a predicate about which we know it holds. For brevity, in our presentation below we indicate the applicability of a method to the goal or to a formula in the knowledge base by an extra condition placed among the preconditions. In the prover implementation, however, this condition is represented differently.

In the current stage of the prover, operator definitions and type information are applied only to ground terms. This might prove to be to restrictive a treatment in more tricky proofs but in the experiments made so far it was sufficient.

Let F be a functor defining a predicate $pred$ of the form shown in formula 4.1. Let D, D_1, ..., D_k be the formal parameters of the definition. When a functor definition is parsed, a method $M = (V, Pre, In, Out, Post)$ is generated from the definition of $pred$ such that:

- $V = \{x_1, \ldots, x_n, D, D_1, \ldots, D_k\}$
- $Pre = \{cond(x_1, \ldots, x_n), D \equiv F[D_1, \ldots, D_k]\}$
- $In = \{ \underset{D}{\mathrm{pred}} [x_1, \ldots, x_n]\}$
- $Out = \{rhs(x_1, \ldots, x_n)\}$
- $Post = \{\}$

Note however, that for an operator definition as in 4.1, we could also generate the method:

- $V = \{x_1, \ldots, x_n, D, D_1, \ldots, D_k\}$
- $Pre = \{cond(x_1, \ldots, x_n), D \equiv F[D_1, \ldots, D_k], \{ \underset{D}{\mathrm{pred}} [x_1, \ldots, x_n]\} \subset kb\}$
- $In = \{ \underset{D}{\mathrm{pred}} [x_1, \ldots, x_n]\}$
- $Out = \{ \underset{D}{\mathrm{pred}} [x_1, \ldots, x_n]\}$
- $Post = \{rhs\}$

which only adds the formula rhs to the knowledge base under the assumption that $cond(x_1, \ldots, x_n)$ and $\underset{D}{\mathrm{pred}} [x_1, \ldots, x_n]$ hold. This treatment is applied for the particular case of the membership operator only, because in this case we want to preserve the information that a symbol is in a certain domain for later usage.

The membership operator enjoys a more detailed treatment, in the sense that various methods are generated for particular forms of the right hand side term of the membership operator definition. Specifically, we have dealt with the case in which the membership operator defines the carrier set of the new domain by means of a *constructor* function. Other examples deal with the case of extending an existing domain with additional operators, while keeping the carrier set of the domain unchanged. These two scenarios of building new domains correspond to different particular forms of the general membership definition of the form 4.1.

If the definition of the membership operator is of the form:

$$\underset{D}{\epsilon}[x] \Leftrightarrow \exists_{\{x_1,\ldots,x_n\}} x \equiv h[x_1,\ldots,x_n] \wedge c(x_1,\ldots x_n). \tag{4.8}$$

then the new domain D is built up over existing domains by means of the constructor h.

For such definitions the following method is generated:

- $V = \{x, x_1, \ldots, x_n, D, D_1, \ldots, D_k\}$
- $Pre = \{D \equiv F[D_1, \ldots, D_k],\ \underset{D}{\epsilon}[h[x_1, \ldots, x_n]] = g\}$
- $In = \{\underset{D}{\epsilon}[h[x_1, \ldots, x_n]]\}$
- $Out = \{c(x_1, \ldots x_n)\}$
- $Post = \{\}$

The method is easily obtained by particularizing 4.8 with $x = h(x_1, \ldots, x_n)$. Although apparently trivial the method proved very useful in proving closure properties. In such proofs, after applying some functional operator definitions to the goal we are left with proving that a term of the form $h(x_1, \ldots, x_n)$ is in a given domain, where h is the "constructor" symbol of the domain.

Functional operators and constant operators are treated analogously. As an example, consider the definition of the functor **CartesianProduct** in Example 2. The method generated for the operator $\otimes$ is the following

Variables: $\{a1, a2, b1, b2, CP, A, B\}$

Preconditions: $\{\}$

Input: $\langle a1, b1\rangle \underset{CP}{\otimes} \langle a2, b2\rangle$

Output: $\langle a1 \underset{A}{\otimes} a2, b1 \underset{B}{\otimes} b2\rangle$

PostConditions: $\{\}$.

4.6.2 Applying Type Knowledge

The methods used for applying the definition of the membership operator in a domain are generated as shown in the previous section. This allows us to expand terms of the form $\underset{D}{\epsilon}[t_1, \ldots, t_k]$ appearing in a proof situation.

In many cases, however, we want to use the membership operator of a domain to rewrite a term which is known to belong to that domain or to gain additional information about that term.

For instance, in the knowledge transport properties we prove in the following chapter we reduce the proof of a predicate or equality in a domain to proving one or more predicates or equalities in the underlying domains in terms of which the "target" domain is defined. One step in performing this reduction is to express the elements of the target domain as combinations of elements in the underlying domains. For instance, given a domain C defined as the **CartesianProduct** of domains A and B, via the functor introduced in Example 2, if we know that if $\underset{C}{\epsilon}[t]$ for some term t, then we can deduce that there exist t_1 and t_2 such that $\underset{A}{\epsilon}[t_1]$ and $\underset{B}{\epsilon}[t_2]$ and $t \equiv \langle t_1, t_2\rangle$.

Similarly, we could define a recursive functor $PseudoNat$ as follows:

$$
\begin{array}{l}
\mathrm{PseudoNat}[] \equiv \\
\mathrm{Functor}[\{PN,\ \epsilon\}, \\
\langle "\epsilon", \forall_{\langle x\rangle} \underset{PN}{\epsilon}[x] \Leftrightarrow (x \equiv o) \vee \left(\exists_{\langle y\rangle}(x \equiv s[y]) \wedge \underset{PN}{\epsilon}[y]\right)\rangle; \\
\quad]
\end{array}
$$

based on the two unary and respectively zero-ary symbols s and o. In this case if $PNat \equiv PseudoNat[]$ for a domain $PNat$, then applying type knowledge for a term t such that $\underset{PNat}{\epsilon}[t]$ would mean inferring that t is either o or there exists t' such that $t \equiv s[t']$ and $\underset{PNat}{\epsilon}[t']$.

As a parenthesis, the functor $PseudoNat$ defined in this example, does not, of course generate domains which are isomorphic with the natural numbers, since axioms like the one stating that zero is not the successor of any natural number are missing and can only be stated in our language by means of domain properties. Additionally, the symbols o and s may have additional properties attached, making the set $\{o, s[o], s[s[o]], \ldots\}$ "collapse" into classes of equal elements.

Methods generated for predicate definitions of the form 4.1 simply add to the knowledge base the right hand side of the equivalence 4.1 properly instantiated on a ground term which is known to be a member of the domain D. More specifically, let F be a functor containing a membership predicate definition of the form shown in 4.1. Let D, D_1, ..., D_k be the formal parameters of the functor definition. A method $M = (V, Pre, In, Out, Post)$ is generated from the definition of $\underset{D}{\epsilon}$ such that:

- $V = \{x, x_1, \ldots, x_n, D, D_1, \ldots, D_k\}$
- $Pre = \{\underset{D}{\epsilon}[x], D \equiv F[D_1, \ldots, D_k]\}$
- $In = \{x\}$
- $Out = \{x\}$
- $Post = \{rhs(x_1, \ldots, x_n)\}$

Hence terms t about which we know that they are in the carrier set of D are not rewritten but additional knowledge is derived from the definition of the membership predicate and added to the knowledge base.

The cases in which the membership predicate contains an existential quantifier as in 4.8 is specially treated. With the notations above, the following method is generated:

- $V = \{x, x_1, \ldots, x_n, D, D_1, \ldots, D_k\}$
- $Pre = \{D \equiv F[D_1, \ldots, D_k], \{\underset{D}{\in}[x]\} \in kb\}$
- $In = \{x\}$
- $Out = \{h(t_1, \ldots, t_n)\}\}$
- $Post = \{c(t_1, \ldots t_n)\}$

Note that in the definition above, t_1, ..., t_n are freshly introduced symbols, which in the body of the method are marked as $Fresh[t_i]$. When a method is applied during a proof, symbols wrapped in the tag $Fresh$ are replaced with new names generated by a global *Theorema* function.

The functor parser also supports membership predicate definitions in which the right hand side of the defining equivalence is a disjunction of "cases", like in the example definition of the **PseudoNat** functor. For definitions of this type a list of methods, with the same input component is generated and when a type knowledge step is applied a case split is performed, corresponding to the various terms of the disjunction.

4.7 Proof Control in the Functor Prover

The current version of the functor prover, which was used for producing the proofs in Chapter 5 does not use the deduction tree data structure and the proof search facilities presented in Chapter 3. The architecture in Chapter 3 was designed and implemented in order to overcome some difficulties encountered when using the existing functor prover. We give an overview, in this section, of the control mechanisms implemented in the existing functor prover in order to give the reader a better understanding of how the proofs in the following chapter were generated automatically.

For proving the knowledge transport properties presented in the next chapter, we needed a mixture of deduction steps of the following kind:

- predicate logic steps,
- application of property and operator definitions and usage of domain definitions,

- simplification steps.

Hence we relied on the existing predicate logic prover and simplification prover and added the specific functor prover steps as well as some support for proof control. (At the time of this writing each of the three provers are being re-engineered to use the proof search mechanisms presented in Chapter 3). The idea we used for the prover interaction was to consider the predicate logic prover as a sort of "master" prover which dictates the general course of the proof by applying natural deduction rules. The functor prover and the simplification prover would interfere at some points during the proof and under certain conditions, perform some specific operations on the goal and knowledge base and return the resulting proof situation. Hence, the functor prover and the simplification prover were both "black-box" provers performing a sequence of steps uninterruptable by any of the other provers participating the proof. The predicate logic prover was written as a collection of *Mathematica* rules, each applying to a goal and knowledge base of a particular form and returning "a trace" of the deduction step performed, with embedded calls to the predicate logic prover on the proof situations resulting after application of the deduction step. When the expression returned by the predicate logic prover after application of one such step was evaluated by *Mathematica* , the predicate logic prover was called recursively (by the built-in evaluation mechanism of Mathematica) on new proof situations. In this manner a hierarchical proof object was produced upon calling the predicate logic prover on an initial goal and knowledge base.

In order to interfere at certain points in this process, a wrapper function, called `ProofStep` was placed in each rule of the predicate logic prover around the new proof situations returned. By default, this wrapper was defined to call the predicate logic prover itself on the arguments it received, thus allowing the prover to function unchanged as a stand-alone prover. If other provers were involved in the proof then by redefining the `ProofStep` wrapper the new provers could be called.

In general the redefined wrapper function would perform the following steps:

- Analyze current goal and knowledge base.

- If some condition on the goal and knowledge base is fulfilled then

  – redefine `ProofStep` and/or

  - call another prover

- otherwise call the predicate logic prover with goal and knowledge base unchanged.

Several predefined `ProofStep` functions were defined and applied successively during a proof. They are discussed in more detail in the following chapter and illustrated with automatically generated proofs. The deduction tree structure used at that time contained only AND-nodes. The choice of the open proof situation to be treated was done automatically by the *Mathematica* mechanism for evaluating expressions and each applied proof step meant a recursive call to `ProofStep`. Heuristic information was passed between successive calls to `ProofStep` by means of local variables in the `Block` construct which has lexical scoping. This ensured that when returning from a branch of the deduction tree, to an upper node, heuristic information would be automatically and correctly discarded. More specifically we used the construct:

```
Block[{variable=variable}
      .
      .
      .
]
```

which ensures that when entering the body of the block during evaluation, the variable `variable` is initialized to the value it had outside the block and after leaving the block the variable is restored to the same value. This construct was used to implement a proof depth limit mechanism and to pass information (e.g. about some proof step already performed) between successive steps of the proof.

This approach at integrating provers proved successfull and we used it to prove automatically several theorems shown in the following chapter.

However, we also encountered difficulties which led us to develop the proof search mechanism presented in Chapter 3. When developing new provers based on basic interacting provers it is cumbersome to keep the overview of the whole functionality of the `ProofStep` wrapper. The wrapper encapsulates both the call to a prover, setting heuristic information and setting the next wrapper to be used. Thus, code reusal in its implementation becomes impossible. It soon became clear that the provers interaction mechanism must be systematized in order to allow developers of provers to quickly implement the desired interaction patterns. This is how the proof strategy language emerged. The proof object used so far contains no capabilities for backtracking unless explicitly implemented in the rules of the basic provers or in the proof step function. The evaluation mechanism of *Mathematica* which is used for expanding the proof object selects one of the applicable provers (rules) and ignores others. The fact that backtracking has to be hard-coded means a mixture of proof control and inference rules implementation which again makes it difficult to modify the proof control mechanism while preserving the inference rules. The addition of OR nodes in the deduction trees and the separation of the proof search mechanism from the implementation of the proof rules addresses this problem. Finally, it turned out that even fine-grain operations performed by some provers depend on subproofs performed by other provers. This is the case in applying definitions of partial functional operators or predicates or in manipulating the universal and existential quantifier ranges and restrictive conditions (see [Buchberger, Windsteiger, 1998]). IF nodes allow provers to express such conditions and delay the call to a separate prover while performing the respective fine-grain operation.

Chapter 5

Proving Knowledge Transport Properties

In this chapter we present our experiment in proving knowledge transport properties. We show how mathematical facts can be formalized using the language constructs introduced in the previous chapter and what success we achieved in proving some knowledge-transport properties. When building a new mathematical domain over some existing ones it is often the case that algebraic properties in the underlying domains are inherited in the newly constructed one. This situation arises, for instance in the construction of the traditional number domains (i.e. in the well known construction of integers over natural numbers, rational numbers over integers, complex numbers over reals).

We considered the proofs in this area of mathematics as a case study for testing the provers interaction concepts developed in Chapter 3. A similar class of problems was considered by [Gehrke, 1996] who used ML functors combined with the Larch Prover for modeling some of the traditional number domains and for reasoning about them. We present the mathematical concepts in traditional mathematical notation as well as their formulation in *Theorema*.

5.1 Mathematical Background

In the development presented in the remainder of this section we deal with the notions of group, ring, field, equivalence relation. We present the definitions of these abstract algebraic structures (cf. [Zariski, Samuel, 1975]) and we show how they can be expressed as domain properties in *Theorema* .

Definition 6 *A set G which is closed under a given multiplication is called a group if the following conditions (group axioms) are satisfied:*

G1. The set G is not empty.

G2. If a, b, $c \in G$, then $(ab)c = a(bc)$.

G3. There exists in G an element e such that

- *(1) For any element a in G, $ea = a$.*
- *(2) For any element a in G there exists an element a' in G such that $a'a = e$.*

Definition 7 *A set R in which 2 binary operations, $+$ (addition) and $\cdot$ (multiplication), are given is called a ring if the following conditions are satisfied:*

R1 R is an abelian group with respect to addition.

R2 If a, b, $c \in R$, the $a(bc) = ab(c)$.

R3 If a, b, $c \in R$, then $a(b+c) = ab+ac$ and $(b+c)a = ba+ca$ (distributive law).

Definition 8 *A ring F is called a field if the following conditions are satisfied:*

F1 F has at least 2 elements.

F2 F has an identity.

F3 Every element of F different from zero has an inverse.

In the construction of rational numbers on top of the rationals we also need the concept of equivalence relation.

Definition 9 *(cf. [Williamson, 1985], pp. 3) A relation ρ on S is*

(1) *Reflexive if for all $x \in S$, $x\rho x$.*

(2) *Symmetric if for all $x, y \in S$, $x\rho y$ implies $y\rho x$.*

(2′) *Antisymmetric if for all $x, y \in S$, $x\rho y$ and $y\rho x$ implies $x = x$.*

(3) *Transitive if for all $x, y, z \in S$, $x\rho y$ and $y\rho z$ implies $x\rho z$.*

A relation ρ that satisfies 1, 2 and 3 is called an equivalence relation. A relation ρ that satisfies 1, $2'$ and 3 i called an order relation.

5.2 Formalization in *Theorema*

The notion of *group* is formalized in the language we have at our disposal as a property of a domain. The binary multiplication operation and its neutral element are made explicit as arguments of the property. The existential quantifier in $G3$ is skolemized and expressed as the unary operator $\ominus$ whose value is the inverse of an element. Hence we express $G3$ as the two formulae "$\oplus$.$\ominus$.left" (for each element there exists an inverse) and "$\ominus$.cls" (the inverse of each element is in the group). Figure 5.1 gives the definition of the *Group* property. The simple language we implemented does not allow us to define new properties as combinations (e.g. conjunctions) of existing ones. The definitions of the ring and field structures must, hence be coded "from scratch" and not depending on the existing *Group* property, as would be more natural. It is, however, possible when calling the prover to specify formulae in the knowledge base as conjunctions of properties, as we will show later on. The definition of the *Ring* property is given in Figure 5.2.

Finally, the fact that a binary relation defined on a given domain is an equivalence can be formalized by means of the property *Equivalence* as shown in Figure 5.4.

$$
\begin{array}{l}
\mathrm{Group}[G, \oplus, o, \ominus] \equiv \\
\langle\langle\text{"}\oplus.assc\text{"}, \\
\forall_{\langle x,y,z\rangle}\Big(\underset{G}{\epsilon}[x] \wedge \underset{G}{\epsilon}[y] \wedge \underset{G}{\epsilon}[z]\Big) \Rightarrow (x\oplus y)\oplus z \equiv x\oplus(y\oplus z)\rangle, \\
\langle\text{"}\oplus.o.left\text{"}, \forall_{\langle x\rangle}\underset{G}{\epsilon}[x] \Rightarrow (o \oplus x \equiv x)\rangle, \\
\langle\text{"}\oplus.\ominus.left\text{"}, \forall_{\langle x\rangle}\underset{G}{\epsilon}[x] \Rightarrow (x \oplus \ominus[x] \equiv o)\rangle, \\
\langle\text{"}\oplus.cls\text{"}, \forall_{\langle x,\, y\rangle}\Big(\underset{G}{\epsilon}[x] \wedge \underset{G}{\epsilon}[y]\Big) \Rightarrow \underset{G}{\epsilon}[x \oplus y]\rangle, \\
\langle\text{"}\ominus.cls\text{"}, \forall_{\langle x\rangle}\Big(\Big(\underset{G}{\epsilon}[x]\Big) \Rightarrow \underset{G}{\epsilon}[\ominus[x]]\Big)\rangle \\
\rangle;
\end{array}
$$

Figure 5.1: Formalization of the notion of *Group*

5.3 Constructing the Complex Numbers

It is well known that the complex numbers can be constructed as ordered pairs over the real numbers (i.e., $\mathbf{C} \cong \mathbf{R} \times \mathbf{R}$ with operations defined in terms of the ones in $\mathbf{R}$ by the rules in Figure 5.5.

In *Theorema* we can express this construction by a functor, taking as argument one domain, say A, which represents the reals and returning a domain C, representing the complex numbers. The definition of this functor is given in Figure 5.6.

Note however, that at the time the functor is defined, no assumptions are made with regard to the algebraic properties of the input domain A. The functor simply encodes *a method* by which we obtain from an arbitrary domain A, another domain C. Once we have defined the functors we are interested in, we can try to prove, in *Theorema*, the following well known fact:

Lemma 3 *If R is a ring w.r.t. the operations $+_R$ and $\times_R$, 0_R is the neutral element of addition ($+_R$) in R, and C is the cartesian product of R with itself, with operations $+_C$, $\times_C$ and 0_C defined by the equalities in 5.5, the*

$$\mathrm{Ring}[R, \oplus, \boldsymbol{o}, \ominus, \otimes] \equiv$$
$$\langle\langle\text{"} \oplus .assc\text{"}, \forall_{\langle x,\ y,\ z\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y] \wedge \underset{R}{\epsilon}[z]\Big) \Rightarrow$$
$$(x \oplus y) \oplus z \equiv x \oplus (y \oplus z)\rangle,$$
$$\langle\text{"} \oplus .o.left\text{"}, \forall_{\langle x\rangle} \underset{R}{\epsilon}[x] \Rightarrow \boldsymbol{o} \oplus x \equiv x\rangle,$$
$$\langle\text{"} \oplus . \ominus .left\text{"}, \forall_{\langle x\rangle} \underset{R}{\epsilon}[x] \Rightarrow x \oplus \ominus[x] \equiv o\rangle,$$
$$\langle\text{"} \oplus .com\text{"}, \forall_{\langle x,\ y\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \Rightarrow (x \oplus y) \equiv (y \oplus x)\rangle,$$
$$\langle\text{"} \otimes .assc\text{"}, \forall_{\langle x,\ y,\ z\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y] \wedge \underset{R}{\epsilon}[z]\Big) \Rightarrow$$
$$(x \otimes y) \otimes z \equiv x \otimes (y \otimes z)\rangle,$$
$$\langle\text{"}distl\text{"}, \forall_{\langle x,y,\ z\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y] \wedge \underset{R}{\epsilon}[z]\Big) \Rightarrow$$
$$(x \oplus y) \otimes z \equiv (x \otimes z) \oplus (y \otimes z)\rangle,$$
$$\langle\text{"}distr\text{"}, \forall_{\langle x,\ y,\ z\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y] \wedge \underset{R}{\epsilon}[z]\Big) \Rightarrow$$
$$z \otimes (x \oplus y) \equiv (z \otimes x) \oplus (z \otimes y)\rangle,$$
$$\langle\text{"} \oplus .cls\text{"}, \forall_{\langle x,\ y\rangle}\Big(\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \Rightarrow \underset{R}{\epsilon}[x \oplus y]\Big)\rangle,$$
$$\langle\text{"} \otimes .cls\text{"}, \forall_{\langle x,\ y\rangle}\Big(\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \Rightarrow \underset{R}{\epsilon}[x \otimes y]\Big)\rangle,$$
$$\langle\text{"} \ominus .cls\text{"}, \forall_{\langle x\rangle}\Big(\Big(\underset{R}{\epsilon}[x]\Big) \Rightarrow \underset{R}{\epsilon}[\ominus[x]]\Big)\rangle$$
$$\rangle;$$

Figure 5.2: Definition of the Ring

$$
\begin{aligned}
&\mathrm{RingProperties}[R,\ \oplus,\ o, \ominus,\ \otimes] \\
&\langle\langle\text{"}x.\otimes.\ominus\text{"},\ \forall_{\langle x,\ y\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \ \Rightarrow\ (x\ \otimes\ \ominus[y]\ \equiv\ (\ominus[x\ \otimes\ y]))\rangle, \\
&\langle\text{"}\ominus.\otimes.x\text{"},\ \forall_{\langle x,\ y\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \Rightarrow ((\ominus[x]\ \otimes\ y) \equiv\ \ominus[x\ \otimes\ y])\rangle, \\
&\langle\text{"}\ominus.x.\oplus.y\text{"}, \\
&\qquad \forall_{\langle x,\ y\rangle}\Big(\underset{R}{\epsilon}[x] \wedge \underset{R}{\epsilon}[y]\Big) \Rightarrow ((\ominus[x\ \oplus\ y]) \equiv\ \ominus[x]\ \oplus\ \ominus[y])\rangle\rangle
\end{aligned}
$$

Figure 5.3: Definition of *RingProperties*

$$
\begin{aligned}
&\mathrm{Equivalence}[H, \approx]\ \equiv \\
&\langle\langle\text{"}refl\text{"},\ \forall_{\langle x\rangle}\ \underset{H}{\epsilon}[x]\ \Rightarrow\ (x\ \approx\ x)\rangle, \\
&\langle\text{"}symm\text{"},\ \forall_{\langle x,\ y\rangle}\Big(\underset{H}{\epsilon}[x]\ \wedge\ \underset{H}{\epsilon}[y]\ \wedge (x\ \approx\ y)\Big) \Rightarrow (y \approx x)\rangle, \\
&\langle\text{"}trans\text{"}, \forall_{\langle x,\ y,\ z\rangle}\Big(\underset{H}{\epsilon}[x] \wedge \underset{H}{\epsilon}[y] \wedge \underset{H}{\epsilon}[z] \wedge (x \approx y) \wedge (y \approx z)\Big) \Rightarrow \\
&(x \approx z)\rangle \\
&\rangle;
\end{aligned}
$$

Figure 5.4: Formalization of the definition of equivalence

$$
\begin{aligned}
0_{\mathbf{C}} &= (0_{\mathbf{R}}, 0_{\mathbf{R}}) \\
(a,b) +_{\mathbf{C}} (b,d) &= (a +_{\mathbf{R}} c, b +_{\mathbf{R}} d) \\
-_{\mathbf{C}}(a,b) &= (-_{\mathbf{R}}a, -_{\mathbf{R}}b) \\
(a,b) *_{\mathbf{C}} (b,d) &= (a *_{\mathbf{R}} c +_{\mathbf{R}} (-_{\mathbf{R}}(b *_{\mathbf{R}} d)), (a *_{\mathbf{R}} d) +_{\mathbf{R}} (b *_{\mathbf{R}} c))
\end{aligned}
$$

Figure 5.5: Constructing the Complex Numbers

$$
\begin{aligned}
&\text{Complexification}[A] \;\equiv \\
&\text{Functor}\,[\langle C, \epsilon, \oplus, o, \ominus, \otimes\rangle, \\
&\langle\text{"}\epsilon\text{"}, \forall_{\langle x\rangle} \underset{C}{\epsilon}\,[x] \;\Leftrightarrow\; \exists_{\langle xr, xi\rangle} \underset{A}{\epsilon}\,[xr] \wedge \underset{A}{\epsilon}\,[xi] \wedge \\
&(x \;\equiv\; \langle xr,\; xi\rangle\,)\rangle; \\
&\langle\text{"}\oplus\text{"}, \forall_{\langle x1,\,y1,\,x2,\,y2\rangle}\langle x1, y1\rangle \underset{C}{\oplus} \langle x2,\; y2\rangle \equiv \langle x1 \underset{A}{\oplus} x2,\; y1 \underset{A}{\oplus} y2\rangle\rangle; \\
&\langle\text{"}o\text{"},\; \underset{C}{\mathrm{o}} \;\equiv \langle\; \underset{A}{\mathrm{o}}\;,\; \underset{A}{\mathrm{o}}\;\rangle\rangle; \\
&\langle\text{"}\ominus\text{"}, \forall_{\langle x,\,y\rangle} \underset{C}{\ominus}\,[\langle x,\; y\rangle] \;\equiv\; \langle\; \underset{A}{\ominus}\,[x],\; \underset{A}{\ominus}\,[y]\rangle\rangle; \\
&\langle\text{"}\otimes\text{"}, \forall_{\langle x1,\,y1,\,x2,\,y2\rangle}\langle x1, y1\rangle \underset{C}{\otimes} \langle x2,\; y2\rangle \equiv \\
&\langle\; \Big(x1 \underset{A}{\otimes} x2\Big) \underset{A}{\oplus} \underset{A}{\ominus}\,\Big[\Big(y1 \underset{A}{\otimes} y2\Big)\Big],\; \Big(x1 \underset{A}{\otimes} y2\Big) \underset{A}{\oplus} \Big(y1 \underset{A}{\otimes} x2\Big)\rangle\;\rangle; \\
&\;];
\end{aligned}
$$

Figure 5.6: Definition of the functor *Complexification*

$$\text{FunctorProver}[\langle\text{"}F\text{"}, \Big(\text{Ring}[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}] \wedge$$

$$\text{RingProperties}[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}] \wedge$$

$$(C \equiv \text{Complexification}[A])\Big)$$

$$\Rightarrow$$

$$\text{Ring}[C, \underset{C}{\oplus}, \underset{C}{\text{o}}, \underset{C}{\ominus}, \underset{C}{\otimes}]\rangle,$$

$$\langle\rangle];$$

Figure 5.7: Calling the functor prover

R is a ring with respect to $+_R$ and $\times_R$ and 0_R is the neutral element in R w.r.t addition.

The corresponding formulation in *Theorema* is in Figure 5.7. We call the functor prover with an empty list of assumptions and ask it to prove an implication. The property *RingProperties* appearing among the assumptions is defined in Figure 5.3. It can be proved, by simplification that if a domain is a ring, (i.e. satisfies the property *Ring*) then it also satisfies the property *RingProperties*). This additional knowledge is needed during the proof of the fact that the complexification of a ring is a ring.

In the remainder of this section we present parts of the automatically generated proof of the lemma stated above. We explain what provers perform each part of the proof and we link the deduction steps performed by the functor prover to the inference rules given in Chapter 4.

5.3.1 Applying Property Definitions

The first part of the proof is performed by the predicate logic prover and the functor prover. The predicate logic prover applies the deduction rule to the formula to be proven, which is an implication. When the goal formula becomes atomic with outermost symbol the name of the property, the functor prover applies the definition of the property (according to the inference rules described in Section 4.7).

We show (F)

$\text{Ring}\,[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}]\wedge$
$RingProperties[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}]\wedge$
$(C \equiv \text{Complexification}[A]) \Rightarrow$
$\text{Ring}\,[C, \underset{C}{\oplus}, \underset{C}{\text{o}}, \underset{C}{\ominus}, \underset{C}{\otimes}]$.

We prove (F) by the deduction rule. We assume (F.H)

$\text{Ring}\,[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}]\wedge$
$\text{RingProperties}\,[A, \underset{A}{\oplus}, \underset{A}{\text{o}}, \underset{A}{\ominus}, \underset{A}{\otimes}]\wedge$
$(C \equiv \text{Complexification}[A])$

and show (F.C)

$\text{Ring}\,[C, \underset{C}{\oplus}, \underset{C}{\text{o}}, \underset{C}{\ominus}, \underset{C}{\otimes}]$.

We expand the definition of

$$\text{Ring}\,[C, \underset{C}{\oplus}, \underset{C}{\text{o}}, \underset{C}{\ominus}, \underset{C}{\otimes}]$$

Thus we have to prove (F.C)

$\langle\text{"}\oplus.assc\text{"}, \forall_{\langle x,y,z\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \wedge \underset{C}{\epsilon}[z] \Rightarrow$
$\Big(\Big(x \underset{C}{\oplus} y\Big) \underset{C}{\oplus} z\Big) \equiv \Big(x \underset{C}{\oplus} \Big(y \underset{C}{\oplus} z\Big)\Big)\rangle\wedge$
$\langle\text{"}\oplus.o.left\text{"}, \forall_{\langle x\rangle} \underset{C}{\epsilon}[x] \Rightarrow \Big(\underset{C}{\text{o}} \underset{C}{\oplus} x\Big) \equiv x\rangle\wedge$
$\langle\text{"}\oplus.\ominus.left\text{"}, \forall_{\langle x\rangle} \underset{C}{\epsilon}[x] \Rightarrow \Big(x \underset{C}{\oplus} \underset{C}{\ominus}[x]\Big) \equiv o_C\rangle\wedge$
$\langle\text{"}\oplus.com\text{"}, \forall_{\langle x,y\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \Rightarrow \Big(x \underset{C}{\oplus} y\Big) \equiv \Big(y \underset{C}{\oplus} x\Big)\rangle\wedge$
$\langle\text{"}\otimes.assc\text{"}, \forall_{\langle x,y,z\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \wedge \underset{C}{\epsilon}[z] \Rightarrow$

$$\Bigl(\bigl(x \underset{C}{\otimes} y\bigr) \underset{C}{\otimes} z\Bigr) \equiv \Bigl(x \underset{C}{\otimes} \bigl(y \underset{C}{\otimes} z\bigr)\Bigr)\rangle\wedge$$

$$\langle\text{"}distl\text{"}, \forall_{\langle x,y,z\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \wedge \underset{C}{\epsilon}[z] \Rightarrow$$

$$\Bigl(\bigl(x \underset{C}{\oplus} y\bigr) \underset{C}{\otimes} z\Bigr) \equiv \Bigl(\bigl(x \underset{C}{\otimes} z\bigr) \underset{C}{\oplus} \bigl(y \underset{C}{\otimes} z\bigr)\Bigr)\rangle\wedge$$

$$\langle\text{"}distr\text{"}, \forall_{\langle x,y,z\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \wedge \underset{C}{\epsilon}[z] \Rightarrow$$

$$\Bigl(z \underset{C}{\otimes} \bigl(x \underset{C}{\oplus} y\bigr)\Bigr) \equiv \Bigl(\bigl(z \underset{C}{\otimes} x\bigr) \underset{C}{\oplus} \bigl(z \underset{C}{\otimes} y\bigr)\Bigr)\rangle\wedge$$

$$\langle\text{"}\oplus.cls\text{"},\ \forall_{\langle x,\ y\rangle}\Bigl(\Bigl(\underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y]\Bigr) \Rightarrow \underset{C}{\epsilon}[x \underset{C}{\oplus} y]\Bigr)\rangle\wedge$$

$$\langle\text{"}\otimes.cls\text{"},\ \forall_{\langle x,\ y\rangle}\Bigl(\Bigl(\underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y]\Bigr) \Rightarrow \underset{C}{\epsilon}[x \underset{C}{\otimes} y]\Bigr)\rangle\wedge$$

$$\langle\text{"}\ominus.cls\text{"},\ \forall_{\langle x\rangle}\Bigl(\Bigl(\underset{C}{\epsilon}[x]\Bigr) \Rightarrow \underset{C}{\epsilon}[\underset{C}{\ominus}[x]]\Bigr)\rangle$$

under the assumptions

(F.H.1) $\text{Ring}[A,\ \underset{A}{\oplus},\ \underset{A}{\text{o}},\ \underset{A}{\ominus},\ \underset{A}{\otimes}]$,

(F.H.2) $\text{RingProperties}[A,\ \underset{A}{\oplus},\ \underset{A}{\text{o}},\ \underset{A}{\ominus},\ \underset{A}{\otimes}]$,

(F.H.3) $C \equiv \text{Complexification}[A]$.

We prove the individual conjunctive parts of (F.C).

The rest of the notebook contains the proofs of the 10 ring axioms, each in a separate cell.

The individual proofs of the ring axioms fall into two proving "patterns", but are similar in as far as the formula to be proven in the domain constructed by *Complexification* is reduced to proving a formula in the underlying domain, by successively applying operators' definitions. The proof of the formula in the underlying domain then is done either by the predicate logic prover (as in the case of closure properties) or by simplification (as in the case of the other ring axioms), or by a combination of both.

We will therefore illustrate the prover interaction patterns with one example of a closure proof, namely the proof of the formula labeled "$\otimes.cls$" above and one example of a proof of an algebraic property, i.e. the proof of the formula labeled "$\otimes.assc$".

For each of these examples we will show the parts performed by each prover in the automatically generated proof and we will then explain how the desired strategy can be implemented using the proof strategy language presented in Section 3.7.1.

5.3.2 Proving Closure of Multiplication

The formula to be proven is:
$\langle$" $F.C.\otimes.cls$", $\forall_{\langle x,\, y\rangle}\Big(\big(\underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y]\big) \Rightarrow \underset{C}{\epsilon}[x \underset{C}{\otimes} y]\Big)\rangle$
The proof starts with the application of the deduction rule to the goal formula and the instantiation of the universally quantified variables to arbitrary but fix constants, by the predicate logic prover:

For proving (F.C.$\otimes$.cls) we prove, for arbitrary but fixed values, (F.C.$\otimes$.cls')
$\underset{C}{\epsilon}[x_0] \wedge \underset{C}{\epsilon}[y_0] \Rightarrow \underset{C}{\epsilon}[\big(x_0 \underset{C}{\otimes} y_0\big)]$.
We prove (F.C.$\otimes$.cls') by the deduction rule.
We assume (F.C.$\otimes$.cls'.H)
$\underset{C}{\epsilon}[x_0] \wedge \underset{C}{\epsilon}[y_0]$
and show (F.C.$\otimes$.cls'.C)
$\underset{C}{\epsilon}[\big(x_0 \underset{C}{\otimes} y_0\big)]$.

At this point the goal became an atomic formula whose outermost symbol is $\underset{C}{\epsilon}$. On goals of this form the functor prover tries to apply the definition of $\underset{C}{\otimes}$. The method it tries to apply, generated from the definition of the functor *Complexification* has the following components:

V is the list $\{A, C, x1, y1, x2, y2\}$,

Pre is $C \equiv Complexification[A]$,

In is the term $\langle x1, y1\rangle \underset{C}{\otimes} \langle x2, y2\rangle$,

Out is $\langle \left(x1 \underset{A}{\otimes} x2\right) \underset{A}{\oplus} \underset{A}{\ominus} [\left(y1 \underset{A}{\otimes} y2\right)], \left(x1 \underset{A}{\otimes} y2\right) \underset{A}{\oplus} \left(y1 \underset{A}{\otimes} x2\right)\rangle \rangle$,

$Post$ is the empty list.

Since the goal does not match the term in the input of the method with variables A, C, $x1$, $x2$, $y1$, $y2$ the method is not applicable to the current form of the goal.

Hence the prover tries to apply type knowledge to the operands of $\underset{C}{\otimes}$, namely x_0 and y_0. Both operands are known to be in the carrier set of C, hence they can be expressed as $\langle xr_1, xi_1\rangle$ and respectively $\langle xr_2, xi_2\rangle$ with xr_1, xi_1, xr_2, xi_2 fresh variables in the carrier set of A. After this step the input field of the method above matches the goal, so the definition of $\underset{C}{\otimes}$ is applied. Finally the definition of $\underset{C}{\epsilon}$ becomes applicable. The corresponding method is given below.

V is the list $\{A, C, xr, xi\}$,

Pre is $C \equiv Complexification[A]$,

In is the term $\underset{C}{\epsilon}[\langle xr, xi\rangle]$,

Out is $\underset{A}{\epsilon}[xr] \wedge \underset{A}{\epsilon}[xi]$,

$Post$ is the empty list.

After applying it the goal becomes a conjunction containing only variables and operators in the underlying domain A.

The part of the proof performing the steps we described is given below. Normally, after applying a definition to the goal and knowledge base, the functor prover displays the new goal and the entire knowledge base obtained. For proofs in notebook format this is no hindrance since the cells holding formulae in the knowledge base have been grouped and can be closed and opened together as desired. However, in the text below, in order to help the reader keep the overview of the proof we only display the parts of the knowledge base which changed during the latest deduction step applied. The fact that only a part of the knowledge base is displayed is signaled by a row of vertical dots.

Trying to apply the definition of $\underset{C}{\otimes}$ to

(F.C.$\otimes$.cls'.C) $\underset{C}{\epsilon}$ $[\left(x_0 \underset{C}{\otimes} y_0\right)]$.

Since x_0 is a member of the carrier set of C we can write it as $\langle xr_1, xi_1\rangle$.

We show (F.C.$\otimes$.cls'.C.C.ϵ)
$\underset{C}{\epsilon}$ $[\left(\langle xr_1, xi_1\rangle \underset{C}{\otimes} y_0\right)]$
under the assumptions
(F.C.$\otimes$.cls'.H.1) $\underset{C}{\epsilon}$ $[\langle xr_1, xi_1\rangle]$,
(F.C.$\otimes$.cls'.C.1) $\underset{A}{\epsilon}$ $[xr_1]$,
(F.C.$\otimes$.cls'.C.2) $\underset{A}{\epsilon}$ $[xi_1]$.

$\vdots$

Since y_0 is a member of the carrier set of C we can write it as $\langle xr_2, xi_2\rangle$.

We show (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ)
$\underset{C}{\epsilon}$ $[\left(\langle xr_1, xi_1\rangle \underset{C}{\otimes} \langle xr_2, xi_2\rangle\right)]$ under the assumptions
(F.C.$\otimes$.cls'.H.2) $\underset{C}{\epsilon}$ $[\langle xr_2, xi_2\rangle]$,
(F.C.$\otimes$.cls'.C.C.ϵ.1) $\underset{A}{\epsilon}$ $[xr_2]$,
(F.C.$\otimes$.cls'.C.C.ϵ.2) $\underset{A}{\epsilon}$ $[xi_2]$

$\vdots$

We apply the definition of $\underset{C}{\otimes}$ to the goal.

We show (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$)

$$\underset{C}{\epsilon}\ [\langle\Big(\big(xr_1 \underset{A}{\otimes} xr_2\big) \underset{A}{\oplus} \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]\Big),$$
$$\Big(\big(xr_1 \underset{A}{\otimes} xi_2\big) \underset{A}{\oplus} \big(xi_1 \underset{A}{\otimes} xr_2\big)\Big)\rangle]$$

under the assumptions

⋮

We apply the definition of $\underset{C}{\epsilon}$ to the goal.

We show (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ)

$$\underset{A}{\epsilon}\ [\Big(\big(xr_1 \underset{A}{\otimes} xr_2\big) \underset{A}{\oplus} \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]\Big)]\wedge$$
$$\underset{A}{\epsilon}\ [\Big(\big(xr_1 \underset{A}{\otimes} xi_2\big) \underset{A}{\oplus} \big(xi_1 \underset{A}{\otimes} xr_2\big)\Big)]$$

under the assumptions

⋮

From here on the proof is performed by the predicate logic prover using the closure of addition, multiplication and additive inverse in A.

The formula to be proven is a conjunction of ground terms and in order to prove the atomic predicates it can only use (in a backward chaining fashion) the closure of addition and multiplication in A.

Proof of (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.1)

$$\underset{A}{\epsilon}\ [\Big(\big(xr_1 \underset{A}{\otimes} xr_2\big) \underset{A}{\oplus} \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]\Big)]:$$

From (A.$\oplus$.cls) by specialization we obtain

(A.$\oplus$.cls/3)
$$\underset{A}{\epsilon}\ [\big(xr_1 \underset{A}{\otimes} xr_2\big)]\wedge\ \underset{A}{\epsilon}\ [\ \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]]\Rightarrow$$
$$\underset{A}{\epsilon}\ [\Big(\big(xr_1 \underset{A}{\otimes} xr_2\big) \underset{A}{\oplus} \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]\Big)]$$

For proving (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.1), by (A.$\oplus$.cls/3), it suffices to prove

(A.$\oplus$.cls/3.A) $\underset{A}{\epsilon}\ [\big(xr_1 \underset{A}{\otimes} xr_2\big)]\wedge\ \underset{A}{\epsilon}\ [\ \underset{A}{\ominus}[\big(xi_1 \underset{A}{\otimes} xi_2\big)]]$.

We prove the individual conjunctive parts of (A.$\oplus$.cls/3.A):

Proof of (A.$\oplus$.cls/3.A.1) $\underset{A}{\epsilon}\,[\big(xr_1 \underset{A}{\otimes} xr_2\big)]$:

From (A.$\otimes$.cls) by specialization we obtain
(A.$\otimes$.cls/3) $\underset{A}{\epsilon}\,[xr_1] \wedge \underset{A}{\epsilon}\,[xr_2] \Rightarrow \underset{A}{\epsilon}\,[\big(xr_1 \underset{A}{\otimes} xr_2\big)]$.

For proving (A.$\oplus$.cls/3.A.1), by (A.$\otimes$.cls/3), it suffices to prove
(A.$\oplus$.cls/3.A) $\underset{A}{\epsilon}\,[xr_1] \wedge \underset{A}{\epsilon}\,[xr_2]$.

We prove the individual conjunctive parts of (A.$\oplus$.cls/3.A):

Proof of (A.$\oplus$.cls/3.A.1) $\underset{A}{\epsilon}\,[xr_1]$:

Formula (A.$\oplus$.cls/3.A.1) is true because it is identical to (F.C.$\otimes$.cls'.C.1).

Proof of (A.$\oplus$.cls/3.A.2) $\underset{A}{\epsilon}\,[xr_2]$:

Formula (A.$\oplus$.cls/3.A.2) is true because it is identical to (F.C.$\otimes$.cls'.C.C.ϵ.1).

Proof of (A.$\oplus$.cls/3.A.2) $\underset{A}{\epsilon}\,[\underset{A}{\ominus}\,[\big(xi_1 \underset{A}{\otimes} xi_2\big)]]$:

From (A.$\ominus$.cls) by specialization we obtain
(A.$\ominus$.cls/2) $\underset{A}{\epsilon}\,[\big(xi_1 \underset{A}{\otimes} xi_2\big)] \Rightarrow \underset{A}{\epsilon}\,[\underset{A}{\ominus}\,[\big(xi_1 \underset{A}{\otimes} xi_2\big)]]$.

For proving (A.$\oplus$.cls/3.A.2), by (A.$\ominus$.cls/2), it suffices to prove
(A.$\ominus$.cls/2.A) $\underset{A}{\epsilon}\,[\big(xi_1 \underset{A}{\otimes} xi_2\big)]$.

From (A.$\otimes$.cls) by specialization we obtain
(A.$\otimes$.cls/3) $\underset{A}{\epsilon}\,[xi_1] \wedge \underset{A}{\epsilon}\,[xi_2] \Rightarrow \underset{A}{\epsilon}\,[\big(xi_1 \underset{A}{\otimes} xi_2\big)]$.

For proving (A.$\ominus$.cls/2.A), by (A.$\oplus$.cls/3), it suffices to prove
(A.$\oplus$.cls/3.A) $\underset{A}{\epsilon}\,[xi_1] \wedge \underset{A}{\epsilon}\,[xi_2]$.

We prove the individual conjunctive parts of (A.$\oplus$.cls/3.A):

Proof of (A.$\oplus$.cls/3.A.1) $\underset{A}{\epsilon}\,[xi_1]$:

Formula (A.$\oplus$.cls/3.A.1) is true because it is identical to (F.C.$\otimes$.cls'.C.2).

Proof of (A.$\oplus$.cls/3.A.2) $\underset{A}{\epsilon}\,[xi_2]$:

Formula (A.$\oplus$.cls/3.A.2) is true because it is identical to (F.C.$\otimes$.cls'.C.C.ϵ.2).

Proof of (F.C.$\otimes$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.2)

$\underset{A}{\epsilon}\,[\Big(\Big(xr_1 \underset{A}{\otimes} xi_2\Big) \underset{A}{\oplus} \Big(xi_1 \underset{A}{\otimes} xr_2\Big)\Big)]$:

From (A.⊕.cls) by specialization we obtain

(A.⊕.cls/3) $$\begin{array}{l}\underset{A}{\epsilon}\,[\Big(xr_1 \underset{A}{\otimes} xi_2\Big)] \wedge \underset{A}{\epsilon}\,[\Big(xi_1 \underset{A}{\otimes} xr_2\Big)] \Rightarrow \\ \quad \underset{A}{\epsilon}\,[\Big(\Big(xr_1 \underset{A}{\otimes} xi_2\Big) \underset{A}{\oplus} \Big(xi_1 \underset{A}{\otimes} xr_2\Big)\Big)]\end{array}.$$

For proving (F.C.⊗.cls'.C.C.ϵ.C.ϵ.C.⊗.C.ϵ.2), by (A.⊕.cls/3), it suffices to prove

(A.⊕.cls/3.A) $\underset{A}{\epsilon}\,[\Big(xr_1 \underset{A}{\otimes} xi_2\Big)] \wedge \underset{A}{\epsilon}\,[\Big(xi_1 \underset{A}{\otimes} xr_2\Big)]$.

We prove the individual conjunctive parts of (A.⊕.cls/3.A):

Proof of (A.⊕.cls/3.A.1) $\underset{A}{\epsilon}\,[\Big(xr_1 \underset{A}{\otimes} xi_2\Big)]$:

From (A.⊗.cls) by specialization we obtain

(A.⊗.cls/3) $\underset{A}{\epsilon}\,[xi_1] \wedge \underset{A}{\epsilon}\,[xr_2] \Rightarrow \underset{A}{\epsilon}\,[\Big(xi_1 \underset{A}{\otimes} xr_2\Big)]$.

For proving (A.⊕.cls/3.A.2), by (A.⊗.cls/3), it suffices to prove

(A.⊕.cls/3.A) $\underset{A}{\epsilon}\,[xi_1] \wedge \underset{A}{\epsilon}\,[xr_2]$.

We prove the individual conjunctive parts of (A.⊕.cls/3.A):

Proof of (A.⊕.cls/3.A.1) $\underset{A}{\epsilon}\,[xi_1]$:

Formula (A.⊕.cls/3.A.1) is true because it is identical to (F.C.⊗.cls'.C.2).

Proof of (A.⊕.cls/3.A.2) $\underset{A}{\epsilon}\,[xr_2]$:

Formula (A.⊕.cls/3.A.2) is true because it is identical to (F.C.⊗.cls'.C.C.ϵ.1).

The strategy adopted for this proof was to let the predicate logic prover act as default prover and interfere at the points in the proof where domain property definitions or domain operator definitions become applicable. As shown in Section 4.7 this is currently achieved by redefining the function `ProofStep` appropriately. We will now explain in more detail how this function works and we will examine how the current behavior can be expressed in terms of the proof strategy language introduced in Section 3.7.1.

5.3.3 Using the Function `ProofStep` to Implement a Strategy

The strategy we described briefly above uses 3 `ProofStep` functions, namely `ApplyPropertyDefs`, `ApplyDefsToGoal` and `ApplyDefsToKB`, each taking as argument a goal and a knowledge base and returning a deduction tree.

Recall that `ProofStep` is executed after each call to the predicate logic prover. The function implementing the predicate logic prover is called `PND`. The function implementing the simplification prover is called `Simplifier`. Initially, `ProofStep` is set to `ApplyPropertyDefs`, which is described below.

In: g, kb.

Out: a deduction tree p.

Step 1: If g is an atomic formula with outermost symbol a property name then goto *Step 2*, else goto *Step 3*.

Step 2: Set `ProofStep` to `ApplyDefsToGoal`.
Let g' be the goal obtained from g by replacing the domain property with its definition.
Let t be a trace of the deduction step performed.
Compose p from t and `PND`$[g', kb]$.
Return p.

Step 3: Return `PND`$[g, kb]$.

`ApplyDefsToGoal` performs the following operations:

In: g, kb.

Out: a deduction tree p.

Step 1: Attempt to apply the definition of a domain operator to g. If this succeeds goto *Step 2*, else goto *Step 3*.

Step 2: Let g', kb' be the goal, respectively the knowledge base obtained in *Step 1*. Return `ProofStep`$[g', kb']$.

Step 3: Set `ProofStep` to `ApplyDefsToKB`. Return `PND`$[g, kb]$.

Note that as long as some operator definitions are still applicable to the goal, the function `ApplyDefsToGoal` remains the current `ProofStep` and calls itself recursively. In the sample proof above this corresponds to the part where the definitions of $\underset{C}{\otimes}$ and $\underset{C}{\epsilon}$ were applied to the goal.

So far our strategy was to reduce the initial proof to one in the underlying domain and then call either the simplifier or the predicate logic prover depending on the form of the formula to be proven. So the function `ApplyDefsToKB` comes into play only to prepare this final stage of the proof. Up to this point all domain property definitions in the knowledge base were kept unexpanded.

Before calling one of the two basic provers to finish the proof, `ApplyDefsToKB` expands the domain property definitions. Its description is given below.

In: g, kb.

Out: a deduction tree p.

Step 1: Replace domain properties in the knowledge base with their definition. Let kb' be the knowledge base thus obtained.

Step 2: Set `ProofStep` to `PND`.

Step 3: If g is a formula with equality as outermost symbol, return `Simplifier`$[g, kb']$, otherwise return `PND`$[g, kb']$.

After this step the functor prover will not be applied anymore.

5.3.4 Using the Proof Strategy Language to Implement a Strategy

We want to convince ourselves that the strategy described in the previous section can be expressed in terms of the proof strategy language introduced in Section 3.7.1.

The basic provers participating in the proof are:

- the predicate logic prover (`PND`)
- the simplifier (`Simplifier`)

- the functor prover

The functor prover performs two kinds of deduction steps which we take as separate basic provers:

- applying domain property definitions (`DomPropProver`)
- applying operator definitions (`OpDefProver`)

As shown in the previous section the predicate logic prover is the default prover, a fact which we can express by a proof strategy clause of the form:

`•nonExclusive[True, PND]`

The basic prover `DomPropProver` should apply to proof situations in which the goal is an atomic formula with outermost symbol the name of a property. Let `GoalIsDomProp` be a *Mathematica* function "recognizing" proof situations of this type.

Furthermore, the clause is exclusive, because any successful proof (of the type we design this strategy for) contains a step which applies the definition of domain properties to the goal. Non-exclusive clauses correspond to the case in which there might be other ways to handle the respective proof situations besides the one of calling the prover in the clause.

The proof strategy clause expressing this behavior is

`•exclusive[GoalIsDomProp, DomPropProver]`

Furthermore, we need to apply operator definitions to the goal repeatedly until all terms containing operators in the "target" domain are expressed in terms of operators in the underlying domain. The corresponding clause is:

`•exclusive[GoalHasOps, •repeat[OpDefProver]]` where `GoalHasOps` is a predicate testing for occurrences of the operators of the "target domain" in the goal.

Finally, we want to apply domain property definitions to the knowledge base as a last step before calling one of the two provers `PND` or `Simplifier` to finish the proof in the underlying domain. Let `KBHasOps` be a predicate which returns $True$ on proof situations whose knowledge base contains occurrences of operators of the target domain and $False$ on all other proof situations. The corresponding clause then is:

```
•exclusive[KBHasOps, •repeat[OpDefProver]]
```

We add a non-exclusive clause for the simplification prover and get the following total set of proof strategy clauses:

```
AddClauses[  •nonExclusive[True, PND],
             •nonExclusive[True, Simplifier],
             •exclusive[GoalIsDomProp, DomPropProver],
             •exclusive[GoalHasOp, •repeat[OpDefProver]],
             •exclusive[KBHasDomProp,
                •repeat[DomPropProver]]
]
```

5.3.5 Proving the Associativity of Addition

The subproof of the associativity of addition starts with the application of some rules of predicate logic.

For proving (F.C.$\oplus$.assc) we prove, for arbitrary but fixed values,
(F.C.$\oplus$.assc') $\underset{C}{\epsilon}[x_0] \wedge \underset{C}{\epsilon}[y_0] \wedge \underset{C}{\epsilon}[z_0] \Rightarrow \left(\left(x_0 \underset{C}{\oplus} y_0\right) \underset{C}{\oplus} z_0\right) \equiv \left(x_0 \underset{C}{\oplus} \left(y_0 \underset{C}{\oplus} z_0\right)\right)$.
We prove (F.C.$\oplus$.assc') by the deduction rule.
We assume (F.C.$\oplus$.assc'.H) $\underset{C}{\epsilon}[x_0] \wedge \underset{C}{\epsilon}[y_0] \wedge \underset{C}{\epsilon}[z_0]$ and show

(F.C.$\oplus$.assc'.C) $\left(\left(x_0 \underset{C}{\oplus} y_0\right) \underset{C}{\oplus} z_0\right) \equiv \left(x_0 \underset{C}{\oplus} \left(y_0 \underset{C}{\oplus} z_0\right)\right)$

The goal then becomes a ground equality whose terms contain the operator $\oplus$ in C. The functor prover now expresses the variables x_0, y_0, z_0, appearing as operands of $\underset{C}{\oplus}$ as pairs of elements in A, by applying the membership operator in C. The definitions of operators in C can then be applied.

Trying to apply the definition of $\underset{C}{\oplus}$ to

(F.C.$\oplus$.assc'.C) $\left(\left(x_0 \underset{C}{\oplus} y_0\right) \underset{C}{\oplus} z_0\right) \equiv \left(x_0 \underset{C}{\oplus} \left(y_0 \underset{C}{\oplus} z_0\right)\right)$.

Since y_0 is a member of the carrier set of C we can write it as

$\langle xr_1, xi_1\rangle$.
We show (F.C.$\oplus$.assc'.C.C.ϵ)
$$\left(\left(x_0 \underset{C}{\oplus} \langle xr_1, xi_1\rangle\right) \underset{C}{\oplus} z_0\right) \equiv \left(x_0 \underset{C}{\oplus} \left(\langle xr_1, xi_1\rangle \underset{C}{\oplus} z_0\right)\right)$$
under the assumptions
(F.C.$\oplus$.assc'.H.2) $\underset{C}{\epsilon}\ [\langle xr_1, xi_1\rangle]$,
(F.C.$\oplus$.assc'.C.1) $\underset{A}{\epsilon}\ [xr_1]$,
(F.C.$\oplus$.assc'.C.2) $\underset{A}{\epsilon}\ [xi_1]$

⋮

Since z_0 is a member of the carrier set of C we can write it as $\langle xr_2, xi_2\rangle$.
We show (F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ)
$$\left(\left(x_0 \underset{C}{\oplus} \langle xr_1, xi_1\rangle\right) \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right) \equiv$$
$$\left(x_0 \underset{C}{\oplus} \left(\langle xr_1, xi_1\rangle \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right)\right)$$
under the assumptions
(F.C.$\oplus$.assc'.H.3) $\underset{C}{\epsilon}\ [\langle xr_2, xi_2\rangle]$,
(F.C.$\oplus$.assc'.C.C.ϵ.1) $\underset{A}{\epsilon}\ [xr_2]$,
(F.C.$\oplus$.assc'.C.C.ϵ.2) $\underset{A}{\epsilon}\ [xi_2]$

⋮

We apply the definition of $\underset{C}{\oplus}$ to the goal. We show
(F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$)
$$\left(\left(x_0 \underset{C}{\oplus} \langle xr_1, xi_1\rangle\right) \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right) \equiv$$
$$\left(x_0 \underset{C}{\oplus} \langle\left(xr_1 \underset{A}{\oplus} xr_2\right), \left(xi_1 \underset{A}{\oplus} xi_2\right)\rangle\right)$$
under the assumptions

⋮

Trying to apply the definition of $\underset{C}{\oplus}$ to

(F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$)

$$\left(\left(x_0 \underset{C}{\oplus} \langle xr_1, xi_1\rangle\right) \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right) \equiv$$

$$\left(x_0 \underset{C}{\oplus} \langle\left(xr_1 \underset{A}{\oplus} xr_2\right), \left(xi_1 \underset{A}{\oplus} xi_2\right)\rangle\right) .$$

Since x_0 is a member of the carrier set of C we can write it as $\langle xr_3, xi_3\rangle$.

We show (F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ)

$$\left(\left(\langle xr_3, xi_3\rangle \underset{C}{\oplus} \langle xr_1, xi_1\rangle\right) \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right) \equiv$$

$$\left(\langle xr_3, xi_3\rangle \underset{C}{\oplus} \langle\left(xr_1 \underset{A}{\oplus} xr_2\right), \left(xi_1 \underset{A}{\oplus} xi_2\right)\rangle\right)$$

under the assumptions

(F.C.$\oplus$.assc'.H.1) $\underset{C}{\epsilon}$ $[\langle xr_3, xi_3\rangle]$,

(F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$.1) $\underset{A}{\epsilon}$ $[xr_3]$,

(F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$.2) $\underset{A}{\epsilon}$ $[xi_3]$

$\vdots$

We apply the definition of $\underset{C}{\oplus}$ to the goal.

We show (F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.C.$\oplus$)

$$\left(\langle\left(xr_3 \underset{A}{\oplus} xr_1\right), \left(xi_3 \underset{A}{\oplus} xi_1\right)\rangle \underset{C}{\oplus} \langle xr_2, xi_2\rangle\right) \equiv$$

$$\left(\langle xr_3, xi_3\rangle \underset{C}{\oplus} \langle\left(xr_1 \underset{A}{\oplus} xr_2\right), \left(xi_1 \underset{A}{\oplus} xi_2\right)\rangle\right)$$

under the assumptions

$\vdots$

We apply the definition of $\underset{C}{\oplus}$ to the goal.

We show (F.C.$\oplus$.assc'.C.C.ϵ.C.ϵ.C.$\oplus$.C.ϵ.C.$\oplus$.C.$\oplus$)

$$\Big(\langle\big(xr_3 \underset{A}{\oplus} xr_1\big), \big(xi_3 \underset{A}{\oplus} xi_1\big)\rangle \underset{C}{\oplus} \langle xr_2, xi_2\rangle\Big) \equiv$$
$$\langle\Big(xr_3 \underset{A}{\oplus} \big(xr_1 \underset{A}{\oplus} xr_2\big)\Big), \Big(xi_3 \underset{A}{\oplus} \big(xi_1 \underset{A}{\oplus} xi_2\big)\Big)\rangle$$

under the assumptions

⋮

We apply the definition of $\underset{C}{\oplus}$ to the goal.

We show (F.C.⊕.assc'.C.C.ϵ.C.ϵ.C.⊕.C.ϵ.C.⊕.C.⊕.C.⊕)

$$\langle\Big(\big(xr_3 \underset{A}{\oplus} xr_1\big) \underset{A}{\oplus} xr_2\Big), \Big(\big(xi_3 \underset{A}{\oplus} xi_1\big) \underset{A}{\oplus} xi_2\Big)\rangle \equiv$$
$$\langle\Big(xr_3 \underset{A}{\oplus} \big(xr_1 \underset{A}{\oplus} xr_2\big)\Big), \Big(xi_3 \underset{A}{\oplus} \big(xi_1 \underset{A}{\oplus} xi_2\big)\Big)\rangle$$

under the assumptions

⋮

The goal finally becomes a ground equality in the underlying domain A. At this point the simplification prover is called and proves the equality by reducing the left and right hand side to the same normal form. The simplification prover receives as hypotheses only the properties of the underlying domain A, in our case the formulae in the domain properties *Ring* and *Ring-Properties*, properly instantiated.

For associative and commutative operators the simplification prover uses a flattening technique which transforms, e.g. a term $b \operatorname{op} (a \operatorname{op} c)$ into $a \operatorname{op} b \operatorname{op} c$ if op is known to be commutative and associative. See [Buchberger, Marin, 1997] for more details.

A proof by simplification works.

Simplification of the lhs term:

$$\langle\Big(\big(xr_3 \underset{A}{\oplus} xr_1\big) \underset{A}{\oplus} xr_2\Big), \Big(\big(xi_3 \underset{A}{\oplus} xi_1\big) \underset{A}{\oplus} xi_2\Big)\rangle \quad = \quad \text{by}$$

(Ass.A[⊕]) $\quad \langle(xr_3 \underset{A}{\oplus} xr_1 \underset{A}{\oplus} xr_2), \Big(\big(xi_3 \underset{A}{\oplus} xi_1\big) \underset{A}{\oplus} xi_2\Big)\rangle \quad =$

by (Ass.A[⊕]) $\quad \langle(xr_3 \underset{A}{\oplus} xr_1 \underset{A}{\oplus} xr_2), (xi_3 \underset{A}{\oplus} xi_1 \underset{A}{\oplus} xi_2)\rangle \;\rfloor$

Simplification of the rhs term:
$\langle \left(xr_3 \underset{A}{\oplus} \left(xr_1 \underset{A}{\oplus} xr_2\right)\right), \left(xi_3 \underset{A}{\oplus} \left(xi_1 \underset{A}{\oplus} xi_2\right)\right)\rangle \quad = \quad$ by

(Ass.A[$\oplus$]) $\quad \langle (xr_3 \underset{A}{\oplus} xr_1 \underset{A}{\oplus} xr_2), \left(xi_3 \underset{A}{\oplus} \left(xi_1 \underset{A}{\oplus} xi_2\right)\right)\rangle \quad =$

by (Ass.A[$\oplus$]) $\quad \langle (xr_3 \underset{A}{\oplus} xr_1 \underset{A}{\oplus} xr_2), (xi_3 \underset{A}{\oplus} xi_1 \underset{A}{\oplus} xi_2)\rangle \;\rfloor$

The remaining ring axioms can be proven in a similar manner.

5.4 Towards Constructing the Rational Numbers

When constructing the rational numbers we first consider the set $\mathbf{Z} \otimes \mathbf{Z}$, and then factor it with respect to a certain equivalence relation.

Addition, multiplication, and their inverse operations in $\mathbf{Z} \otimes \mathbf{Z}$ are defined based on the corresponding operations in $\mathbf{Z}$. As in the case of the complex numbers, we capture the construction of $\mathbf{Z} \otimes \mathbf{Z}$ through a functor, called *PreRats*, whose definition is given in Figure 5.8.

As for the *Complexification* functor we can prove that if the underlying domain A is a ring w.r.t. addition and multiplication then the domain R with operations defined as shown is also a ring. The proofs are similar to the ones shown in the previous section with one slight complication. We defined the carrier set of the domain R as $\mathbf{Z} \otimes \mathbf{Z}^*$, hence in all proofs we must handle the condition that for pairs of elements in the carrier set of R the second element of the pair is different from $\underset{A}{\mathrm{o}}$. When proving closure properties, we have to prove that a certain ground term in the carrier set of A is different from $\underset{A}{\mathrm{o}}$ in case the constants appearing in the ground term are different from $\underset{A}{\mathrm{o}}$. We see, after the first failed closure proof, that the underlying domain A must be an integral domain in order for the desired closure properties to hold in R.

As an illustration (and for comparison with the previous section) we give the successful proof of the closure of multiplication in the domain constructed by the functor *PreRats*.

As prerequisites assume we have defined the closure property, say `Closed`

$$
\begin{aligned}
&\mathrm{PreRats}[A] \;\equiv \\
&\mathrm{Functor}\,[\langle R,\ \epsilon,\ \oplus,\ \otimes,\ \approx,\ inv\rangle, \\
&\langle ''\epsilon'', \forall_{\langle x\rangle} \underset{R}{\epsilon}\ [x] \;\Leftrightarrow\; \exists_{\langle xr,xi\rangle}\Big(\ \underset{A}{\epsilon}\ [xr] \wedge \underset{A}{\epsilon}\ [xi] \wedge \\
&(\ x \;\equiv\; \langle xr,\ xi\rangle) \wedge \Big(not\ @\ \Big(xi \;\equiv\; o_A\ \Big)\Big)\Big)\rangle; \\
&\langle ''\oplus'', \forall_{\langle x1,\ y1,\ x2,\ y2\rangle}\langle x1, y1\rangle \underset{R}{\oplus}\ \langle x2,\ y2\rangle \equiv \\
&\langle\Big(x1 \underset{A}{\otimes}\ y2\Big) \underset{A}{\oplus}\ \Big(y1 \underset{A}{\otimes}\ x2\Big)\ ,\ y1 \underset{A}{\otimes}\ y2\rangle\rangle; \\
&\langle ''\ominus'', \forall_{\langle x1,\ y1\rangle}\mathrm{inv}_R[\langle x1, y1\rangle] \equiv \langle\ \underset{A}{\ominus}\ [x1],\ y1\rangle\rangle; \\
&\langle ''\otimes'', \forall_{\langle x1,\ y1,\ x2,\ y2\rangle}\langle x1, y1\rangle \underset{R}{\otimes}\ \langle x2,\ y2\rangle \equiv \\
&\langle\ \Big(x1 \underset{A}{\otimes}\ x2\Big),\ \Big(y1 \underset{A}{\otimes} y2\Big)\rangle\ \rangle; \\
&\langle ''\approx'', \forall_{\langle x1,\ y1,\ x2,\ y2\rangle}(\langle x1, y1\rangle \underset{R}{\approx}\ \langle x2,\ y2\rangle) \Leftrightarrow \\
&\Big(\Big(x1 \underset{A}{\otimes}\ y2\Big) \;\equiv\; \Big(\ x2 \underset{A}{\otimes}\ y1\Big)\Big)\ \rangle; \\
&\ ]
\end{aligned}
$$

Figure 5.8: Towards Constructing the Rational Numbers

$$\text{IntegralDomain}[A,\ \otimes,\ o]\ \equiv$$
$$\langle\langle\text{"}\,intd\text{"},\ \forall_{\langle x,\ y\rangle}\Big(\ \underset{A}{\epsilon}\ [x]\ \wedge\ \ \underset{A}{\epsilon}\ [y]\ \wedge\neg[x\ \equiv o\ ]\ \wedge$$
$$\neg[y\ \equiv\ o]\Big)\ \Rightarrow\neg[(x\otimes\ y)\ \equiv\ o]\ \rangle\rangle;$$

Figure 5.9: The property *Integral Domain*

with arguments a domain and a binary operation. We also need a property expressing the fact that A is an integral domain, given in Figure 5.9.

The automatically generated proof is given in the Appendix.

The relation $\underset{R}{\approx}$ is the one used in factoring $\mathbf{Z}\otimes\mathbf{Z}$ to obtain $\mathbf{Q}$. A problem arising naturally is to prove that the relation $\approx$ defined in the functor *PreRats* is an equivalence. The proofs of reflexivity and symmetry work in the usual manner, by applying the definitions of the operators as we have done in the previous proofs and then calling the simplifier to prove the resulting equality in the underlying domain.

Our prover fails to prove transitivity of $\approx$. We discuss briefly the proof as an example of a situation where user-interaction or proof search would be needed in order to give the prover the "right" idea.

After applying operator definitions as before, the problem is reduced to proving the equality:

$$a\otimes d\equiv b\otimes c$$

under the assumptions:

AC. $\otimes$ is associative and commutative,

T1. $a\otimes f\equiv b\otimes e$

T2. $c\otimes f\equiv d\otimes e$

Can. For all x, y, z , if $z\not\equiv o$ then $x\otimes z=y\otimes z$ if and only if $x=y$.

NZ. $b\not\equiv o,\ d\not\equiv o,\ f\not\equiv o$

The idea is to first prove

$$(a\otimes d)\otimes f\equiv(b\otimes c)\otimes f \tag{5.1}$$

and then use (Can) in order to obtain the desired equality. The proof of 5.1 follows easily by applying the (AC), (T1) and (T2).

In the current version of the prover the proof fails for the following reason: the subproof shown above is passed to the predicate logic prover. The goal is atomic and cannot be decomposed further (the predicate logic prover does not handle equality) so the prover tries to match it with one of the assumptions. It successfully matches it with assumption (Can) and reduces the problem to proving that

$$(a \otimes d) \otimes b \equiv (b \otimes c) \otimes b \tag{5.2}$$

The simplification prover is then called on the resulting goal and fails.

Hence, the predicate logic prover chose the right assumption to use, but instantiated z in (Can) with the wrong value, b. The choice of z leading to a proof would have been f.

One possibility for handling such a proof would be user-interaction. Let us rather look at the possibility to fully automate the proof by means of the search mechanism introduced in Chapter 3. First note that there are only a few choices (among the constants) for instantiating the assumption (Can). Since the constant of choice must be different from o, the only candidates are b, d and f.

Using the new deduction tree format, the predicate logic prover, could, when having to instantiate a variable z in formula (Can), return and OR node having as descendants proof situation nodes which correspond to the possible choices of constants for z. Figure 5.10 shows a partial deduction tree constructed along these ideas. The nodes are labeled with the name of the prover, the type of the node and the current goal.

On each of the proof situations resulting after instantiation, the simplifier is applied. Let us examine first one of the failing branches, e.g. (1). The simplifier returns a new equality obtained after some simplification steps. On the resulting goal, the predicate logic prover can again apply formula (Can), producing three branches, and so on. A similar subtree is built on branch (2).

On branch (3) however, the proof succeeds after the call to the simplification prover. The essential observation is that although the unsuccessful paths can potentially expand infinitely, the successful one is quite short. Hence, in an acceptable amount of time, a breadth-first search would find a proof. This

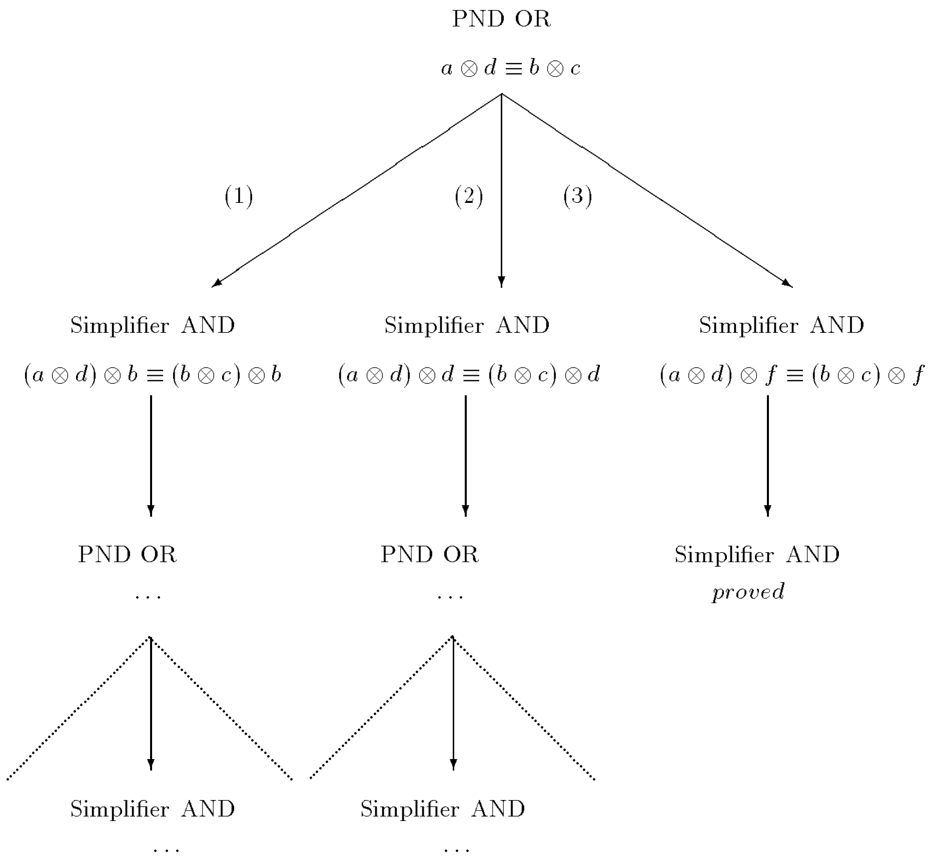

Figure 5.10: Partial Deduction Tree

is a typical situation, for which the search mechanism in Chapter 3 was intended. The proof necessits an invention step but the invention is not an ingenious one and can easily be found by trial and error among the small set of constants which appear in the knowledge base. After instantiation the path corresponding to a successful proof in the deduction tree is short.

Of course, more ingenious choices of terms will necessit user-interaction but the class of "trivial" proofs which we can handle by using the proof search mechanism in Chapter 3 certainly grows compared to the present situation.

Bibliography

[Benzmüller *et al.*, 1997] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMega: Towards a mathematical assistant. In *Proceedings of the 16th Conference on Automated Deduction*, 1997.

[Berndt, 1985] B.C. Berndt. *Ramanujan's Notebooks, Part I.* Springer-Verlag, 1985.

[Boyer, Moore, 1979] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[Buchberger *et al.*, 1997a] B. Buchberger, T. Ida, and D. Vasaru, editors. *Proceedings of the First International Theorema Whorkshop*, Castle of Hagenberg, Austria, jun 1997. Also published as RISC Technical Report No. 97-20.

[Buchberger *et al.*, 1997b] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the *Theorema* project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation), Maui, Hawaii*, jul 1997.

[Buchberger *et al.*, 1997c] B. Buchberger, T. Jebelean, and D. Vasaru. *Theorema*: A system for formal and scientific training in natural language presentation. Technical Report 97-34, Research Institute for Symbolic Computation, 1997. To appear in Proceedings of ED-MEDIA 98, June 20-25, Freiburg, Germany.

[Buchberger, Jebelean, 1997] B. Buchberger and T. Jebelean. The predicate logic prover. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Buchberger, Kriftner, 1997] B. Buchberger and F. Kriftner. *Theorema*: The language. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Buchberger, Lichtenberger, 1981] B. Buchberger and F. Lichtenberger. *Mathematik für Informatiker I.* Springer-Verlag Berlin Heidelberg New York, 1981.

[Buchberger, Marin, 1997] B. Buchberger and M. Marin. Proving by simplification. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Buchberger, Vasaru, 1997] B. Buchberger and D. Vasaru. The induction prover for lists. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Buchberger, Windsteiger, 1998] B. Buchberger and W. Windsteiger. The *Theorema* language: Implementing object- and meta-level usage of symbols. Technical Report 98-10, Research Institute for Symbolic Computation, 1998. Presented at Calculemus-98, Eindhoven, July 13-15, 1998.

[Buchberger, 1993] B. Buchberger. *Mathematica*: Doing mathematics by computer? In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 1–29. Springer Wien - New York, 1993. Invited talk at DISCO 93, Gmunden, Austria.

[Buchberger, 1996a] B. Buchberger. *Mathematica* as a rewrite language. In T. Ida, editor, *The Second Fuji International Workshop on Functional and Logic Programming*, Nov. 1-4, Shonan Village, Japan, 1996. Telos Publishing. Also published as RISC Report 96-40.

[Buchberger, 1996b] B. Buchberger. Mathematische Software-Systeme: Die Zukunft. In *Informatik-Spektrum*. Springer Heidelberg, 1996.

[Buchberger, 1996c] B. Buchberger. Symbolic computation: Computer algebra and logic. In *Frontiers in Combining Systems*, volume 40 of *Applied Logics Series*, pages 193–220. Kluwer Academic Publishers, 1996.

[Buchberger, 1996d] B. Buchberger. Using *Mathematica* for doing simple mathematical proofs. In *4th International Mathematica Users' Conference, Tokyo*. Wolfram Media Publishing, 1996.

[Buchberger, 1997] B. Buchberger. The *Theorema* prover for equalities over natural numbers. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Bundy *et al.*, 1990] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In Mark E. Stickel, editor, *Proceedings of the 10th CADE*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648. Springer Verlag, Berlin, Germany, 1990.

[Bundy *et al.*, 1992] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smail. Rippling: A heuristic for guiding inductive proofs. Technical Report DAI No. 567, Department of AI, University of Edinburgh, February 1992. Submitted to the Artifical Intelligence Journal.

[Bundy, 1988] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *Proceedings of the 9th CADE*, volume 310 of *Lecture Notes in Artificial Intelligence*, pages 112–120. Springer Verlag, Berlin, Germany, 1988.

[Caviness, Johnson, 1998] B.F. Caviness and J.R. Johnson, editors. *Quantifier Elimination by Cylindrical Algebraic Decomposition.* Texts and Monographs in Symbolic Computation. Springer Wien New-York, 1998.

[Clarke, Zhao, 1993] E. Clarke and X. Zhao. Analytica - a theorem prover for *Mathematica. The Mathematica Journal*, 3:56–71, 1993.

[Clarke, Zhao, 1994] E. Clarke and X. Zhao. Combining symbolic computation and theorem proving: some problems of Ramanujan. Technical Report CMU-CS-94-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[Constable, 1986] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Coquand, Huet, 1988] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Dowek *et al.*, 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The system Coq, v. 5.6. user's guide. Technical report, INRIA, 1991.

[Farmer *et al.*, 1993] W.M. Farmer, J.D. Guttman, and F.J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, oct 1993.

[Fikes, Nilsson, 1971] R.E. Fikes and N.J. Nilsson. STRIPS: a new procedure to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Garland, Guttag, 1991] S.J. Garland and J.V. Guttag. A guide to LP, the larch prover. Technical report, Massachussets Institute of Technology, 1991.

[Gehrke, 1996] W. Gehrke. Complex and hyper-complex numbers: A case study for the combination of algebraic computation and deduction. Technical Report RT-INF-7-96, Dipartamento di Discipline Scientifiche, Universita degli Studi di Roma Tre, Via della Vasca Navale, 84-00146 Roma, Italy, 1996.

[Gordon *et al.*, 1979] M.J.C. Gordon, A. Milner, and C.P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Artificial Intelligence.* Springer Verlag Berlin, Germany, 1979.

[Gordon, 1987] M. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification, and Synthesis.* Kluwer, 1987.

[Hewitt, 1977] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Horn, Smaill, 1990] C. Horn and A. Smaill. Theorem proving with Oyster. Technical Report 214, Department of AI, University of Edinburgh, 1990.

[Huang *et al.*, 1994] X. Huang, M. Kerber, J. Richts, and A. Sehn. Planning mathematical proofs with methods. *Journal of Information Processing and Cybernetics*, 30:277–291, 1994.

[Jackson, 1995] P.B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New-York, 1995. Technical Report TR95-1509.

[Jaskowski, 1934] S. Jaskowski. On the rules of supposition in formal logic. *Studia Logica*, (1), 1934.

[Kapur, Zhang, 1989] D. Kapur and H. Zhang. An overview of RRL(Rewrite Rule Laboratory). In *Proceedings of the 3rd International Conference of Rewriting Techniques and Applications*, volume 335 of *Lecture Notes in Computer Science*, pages 559–563. Springer Verlag, Berlin, Germany, 1989.

[Knuth, Bendix, 1970] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y, 1970.

[Martin-Löf, 1982] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[McAllester, 1987] D.A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1987. Also available as MIT Technical Report 979.

[Nilsson, 1980] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[Nipkow, Paulson, 1992] T. Nipkow and L.C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag LNAI 607.

[Pastre, 1989] D. Pastre. An automatic theorem proving system using knowledge and metaknowledge in mathematics. *Artificial Intelligence*, 38:257–318, 1989.

[Pastre, 1993] D. Pastre. Automated theorem proving in mathematics. *Annals of Mathematics and Artificial Intelligence*, 8(3, 4):425–448, 1993.

[Richards *et al.*, 1994] B.L. Richards, I. Kraan, A. Smaill, and G.A. Wiggins. Mollusc: a general proof development shell for sequent-based logics. In A. Bundy, editor, *Proceedings of the 12th CADE*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 826–830. Springer Verlag, Berlin, Germany, 1994.

[Rudnicki, 1992] P. Rudnicki. An overview of the MIZAR project. Notes to a talk at the workshop on *Types for Proofs and Programs*, June 1992.

[Tomuta, 1997] E. Tomuta. Using functors in organizing proofs. In *Proceedings of the First International Theorema Workshop, Hagenberg, Austria*, jun 1997. RISC-Linz Report Series No. 97-20.

[Trybulec, Blair, 1985] A. Trybulec and H. Blair. Computer assisted reasoning with MIZAR. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, Los Angeles, CA, August 1985. Morgan Kaufmann.

[van Harmelen, 1989] F. van Harmelen. The CLAM proof planner. Technical Report TP-4, Department of AI, University of Edinburgh, 1989.

[Williamson, 1985] S.G. Williamson. *Combinatorics for Computer Science*. Computer Science Press, 1985.

[Wolfram, 1996] S. Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 1996.

[Wos, McCune, 1992] L. Wos and W. McCune. The application of automated reasoning to questions in mathematics and logics. *Artificial Intelligence*, 5(2-4):321–369, 1992.

[Zariski, Samuel, 1975] O. Zariski and P. Samuel. *Commutative Algebra I*. Graduate Texts in Mathematics. Springer Verlag, 1975.

Appendix A

A Sample Proof

We prove (F).

We show (F)

$\mathrm{Closed}\,[A, \underset{A}{\oplus}] \wedge \mathrm{Closed}\,[A, \underset{A}{\otimes}] \wedge$

$\mathrm{IntegralDomain}\,[A, \underset{A}{\otimes}, \underset{A}{\mathrm{o}}] \wedge (C \equiv \mathrm{PreRats}[A]) \Rightarrow$.

$\mathrm{Closed}\,[C, \underset{C}{\otimes}]$

We prove (F) by the deduction rule. We assume (F.H)

$\mathrm{Closed}\,[A, \underset{A}{\oplus}] \wedge \mathrm{Closed}\,[A, \underset{A}{\otimes}] \wedge$

$$\mathrm{IntegralDomain}\,[A, \underset{A}{\otimes}, \underset{A}{\mathrm{o}}] \wedge (C \equiv \mathrm{PreRats}[A])$$

and show (F.C)

$\mathrm{Closed}\,[C, \underset{C}{\otimes}]$.

We expand the definition of

$$\mathrm{Closed}\,[C, \underset{C}{\otimes}]$$

Thus we have to prove (F.C)

$\langle\text{"} \oplus .cls\text{"}, \forall_{\langle x,y\rangle} \underset{C}{\epsilon}[x] \wedge \underset{C}{\epsilon}[y] \Rightarrow \underset{C}{\epsilon}[\big(x \underset{C}{\otimes} y\big)]\rangle$

under the assumptions

(F.H.1) $\mathrm{Closed}\,[A, \underset{A}{\oplus}]$, (F.H.2) Closed $[A, \underset{A}{\otimes}]$,

(F.H.3) IntegralDomain $[A, \underset{A}{\otimes}, \underset{A}{\mathrm{o}}]$,

(F.H.4) $C \equiv \mathrm{PreRats}[A]$.

For proving (F.C.$\oplus$.cls) we prove, for arbitrary but fixed values,

(F.C.$\oplus$.cls') $\underset{C}{\epsilon}[x_1] \wedge \underset{C}{\epsilon}[y_1] \Rightarrow \underset{C}{\epsilon}[\left(x_1 \underset{C}{\otimes} y_1\right)]$.

We prove (F.C.$\oplus$.cls') by the deduction rule.

We assume

(F.C.$\oplus$.cls'.H) $\underset{C}{\epsilon}[x_1] \wedge \underset{C}{\epsilon}[y_1]$

and show

(F.C.$\oplus$.cls'.C) $\underset{C}{\epsilon}[\left(x_1 \underset{C}{\otimes} y_1\right)]$.

Trying to apply the definition of $\underset{C}{\otimes}$ to

(F.C.$\oplus$.cls'.C) $\underset{C}{\epsilon}[\left(x_1 \underset{C}{\otimes} y_1\right)]$.

Since x_1 is a member of the carrier set of C we can write it as $\langle xr_4, xi_4 \rangle$.

We show

(F.C.$\oplus$.cls'.C.C.ϵ) $\underset{C}{\epsilon}[\left(\langle xr_4, xi_4 \rangle \underset{C}{\otimes} y_1\right)]$ under the assumptions

(F.C.$\oplus$.cls'.H.1) $\underset{C}{\epsilon}[\langle xr_4, xi_4 \rangle]$,

(F.C.$\oplus$.cls'.C.1) $\underset{A}{\epsilon}[xr_4]$,

(F.C.$\oplus$.cls'.C.2) $\underset{A}{\epsilon}[xi_4]$,

(F.C.$\oplus$.cls'.C.3) $\neg[xi_4 \equiv \underset{A}{\mathrm{o}}]$

$\vdots$

Since y_1 is a member of the carrier set of C we can write it as $\langle xr_5, xi_5 \rangle$.

We show (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ) $\underset{C}{\epsilon}[\left(\langle xr_4, xi_4 \rangle \underset{C}{\otimes} \langle xr_5, xi_5 \rangle\right)]$

under the assumptions

(F.C.$\oplus$.cls'.H.2) $\underset{C}{\epsilon}[\langle xr_5, xi_5 \rangle]$, (F.C.$\oplus$.cls'.C.C.ϵ.1) $\underset{A}{\epsilon}\ [xr_5]$,

(F.C.$\oplus$.cls'.C.C.ϵ.2) $\underset{A}{\epsilon}\ [xi_5]$,

(F.C.$\oplus$.cls'.C.C.ϵ.3) $\neg[xi_5 \equiv \underset{A}{\mathrm{o}}\ ]$

$\vdots$

We apply the definition of $\underset{C}{\otimes}$ to the goal.

We show

(F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$) $\underset{C}{\epsilon}\ [\langle\left(xr_4 \underset{A}{\otimes} xr_5\right), \left(xi_4 \underset{A}{\otimes} xi_5\right)\rangle]$

under the assumptions

$\vdots$

We apply the definition of $\underset{C}{\epsilon}$ to the goal.

We show (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ)

$\underset{A}{\epsilon}\ [\left(xr_4 \underset{A}{\otimes} xr_5\right)] \wedge \underset{A}{\epsilon}\ [\left(xi_4 \underset{A}{\otimes} xi_5\right)] \wedge \neg[\left(xi_4 \underset{A}{\otimes} xi_5\right) \equiv \underset{A}{\mathrm{o}}\ ]$

under the assumptions

$\vdots$

We prove the individual conjunctive parts of

(F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ):

Proof of

(F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.1) $\underset{A}{\epsilon}\ [\left(xr_4 \underset{A}{\otimes} xr_5\right)]$:

From (A.$\oplus$.cls) by specialization we obtain

(A.$\oplus$.cls/3) $\underset{A}{\epsilon}\ [xr_4] \wedge \underset{A}{\epsilon}\ [xr_5] \Rightarrow \underset{A}{\epsilon}\ [\left(xr_4 \underset{A}{\otimes} xr_5\right)]$.

For proving (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.1), by (A.$\oplus$.cls/3), it suffices to prove (A.$\oplus$.cls/3.A) $\underset{A}{\epsilon}\ [xr_4] \wedge \underset{A}{\epsilon}\ [xr_5]$.

We prove the individual conjunctive parts of (A.$\oplus$.cls/3.A):

Proof of (A.$\oplus$.cls/3.A.1) $\underset{A}{\epsilon}\ [xr_4]$:

Formula (A.$\oplus$.cls/3.A.1) is true because it is identical to (F.C.$\oplus$.cls'.C.1). Proof of (A.$\oplus$.cls/3.A.2) $\underset{A}{\epsilon}\ [xr_5]$:

Formula (A.$\oplus$.cls/3.A.2) is true because it is identical to (F.C.$\oplus$.cls'.C.C.ϵ.1).

Proof of (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.2) $\underset{A}{\epsilon}\,[\big(xi_4 \underset{A}{\otimes} xi_5\big)]$:

From (A.$\oplus$.cls) by specialization we obtain (A.$\oplus$.cls/3)
$\underset{A}{\epsilon}\,[xi_4] \wedge \underset{A}{\epsilon}\,[xi_5] \Rightarrow \underset{A}{\epsilon}\,[\big(xi_4 \underset{A}{\otimes} xi_5\big)]$.

For proving (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.2), by (A.$\oplus$.cls/3), it suffices to prove (A.$\oplus$.cls/3.A) $\underset{A}{\epsilon}\,[xi_4] \wedge \underset{A}{\epsilon}\,[xi_5]$.

We prove the individual conjunctive parts of (A.$\oplus$.cls/3.A):

Proof of (A.$\oplus$.cls/3.A.1) $\underset{A}{\epsilon}\,[xi_4]$:

Formula (A.$\oplus$.cls/3.A.1) is true because it is identical to (F.C.$\oplus$.cls'.C.2).

Proof of (A.$\oplus$.cls/3.A.2) $\underset{A}{\epsilon}\,[xi_5]$:

Formula (A.$\oplus$.cls/3.A.2) is true because it is identical to (F.C.$\oplus$.cls'.C.C.ϵ.2).

Proof of (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.3) $\neg[\big(xi_4 \underset{A}{\otimes} xi_5\big) \equiv \underset{A}{\mathrm{o}}\,]$
:

We prove (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.3) by contradiction.
We assume
(F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.3.pos) $\big(xi_4 \underset{A}{\otimes} xi_5\big) \equiv \underset{A}{\mathrm{o}}$,

and show a contradiction.

From (A.intd) by specialization we obtain

(A.intd/5)
$$\begin{array}{l}\underset{A}{\epsilon}\,[xi_4] \wedge \underset{A}{\epsilon}\,[xi_5] \wedge \neg[xi_4 \equiv \underset{A}{\mathrm{o}}\,] \wedge \neg[xi_5 \equiv \underset{A}{\mathrm{o}}\,] \Rightarrow \\ \quad \neg[\big(xi_4 \underset{A}{\otimes} xi_5\big) \equiv \underset{A}{\mathrm{o}}\,]\end{array}$$
.

From (F.C.$\oplus$.cls'.C.C.ϵ.C.ϵ.C.$\otimes$.C.ϵ.3.pos) and (A.intd/5) we obtain by inverse modus ponens
(A.intd/5.C) $\neg[\,\underset{A}{\epsilon}\,[xi_4] \wedge \underset{A}{\epsilon}\,[xi_5] \wedge \neg[xi_4 \equiv \underset{A}{\mathrm{o}}\,] \wedge \neg[xi_5 \equiv \underset{A}{\mathrm{o}}\,]]$
.

Formula (A.intd/5.C) is simplified to

(A.intd/5.C')
$$\begin{array}{l}\neg[\,\underset{A}{\epsilon}\,[xi_4]] \vee \neg[\,\underset{A}{\epsilon}\,[xi_5]] \vee \neg[\neg[xi_4 \equiv \underset{A}{\mathrm{o}}\,]] \vee \\ \quad \neg[\neg[xi_5 \equiv \underset{A}{\mathrm{o}}\,]]\end{array}$$
.

From (F.C.$\oplus$.cls'.C.2) and (A.intd/5.C') we obtain by resolution

(A.intd/5.C'.C) $\neg[\underset{A}{\epsilon}[xi_5]] \vee \neg[\neg[xi_4 \equiv \underset{A}{\mathrm{o}}]] \vee \neg[\neg[xi_5 \equiv \underset{A}{\mathrm{o}}]]$
.
From (F.C.$\oplus$.cls'.C.3) and (A.intd/5.C'.C) we obtain by resolution
(A.intd/5.C'.C.C) $\neg[\underset{A}{\epsilon}[xi_5]] \vee \neg[\neg[xi_5 \equiv \underset{A}{\mathrm{o}}]]$.
From (F.C.$\oplus$.cls'.C.C.ϵ.2) and (A.intd/5.C'.C.C) we obtain by resolution
(A.intd/5.C'.C.C.C) $\neg[\neg[xi_5 \equiv \underset{A}{\mathrm{o}}]]$.
A contradiction) is proved because (A.intd/5.C'.C.C.C) and (F.C.$\oplus$.cls'.C.C.ϵ.3) are contradictory.