Ω-Ants

A Blackboard Architecture for the Integration of Reasoning Techniques into Proof Planning

Volker Sorge

Dissertation

zur Erlangung des Grades des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Saarbrücken, Dezember 2001

Dekan	Prof. Dr. Rainer Schulze-Pillot-Ziemen, Universität des Saarlandes
Vorsitzender	Prof. Dr. Reinhard Wilhelm, Universität des Saarlandes
1. Gutachter	Prof. Dr. Jörg Siekmann, Universität des Saarlandes
2. Gutachter	Prof. Dr. Jaques Calmet, Universität Karlsruhe
Kolloquium	23.11.2001, Universität des Saarlandes, Saarbrücken

Contents

Abstract

Many automated reasoning systems and techniques have been developed for theorem proving for specific mathematical domains. Automated theorem provers and interactive systems for various calculi as well as proof planners have all had some success in limited areas. However, in many challenging interesting domains there is no single system available that has achieved a degree of reliability such that one can be certain this system can solve all problems for such a domain. Therefore, there have been many attempts at combining systems and reasoning techniques over the last decade. In particular, there have been attempts at integrating homogeneous and heterogeneous theorem provers, incorporating decision procedures and symbolic computation, and parallelization of theorem proving.

This thesis presents both novel way of combining reasoning techniques and also the application of these combined techniques to proof planning for group theory and finite algebra. In particular, we combine interactive and automated reasoning, proof planning and symbolic computation. Our means to achieve this combination is a hierarchical blackboard architecture called Ω-Ants. This architecture was originally developed to support the user in interactive theorem proving by searching for possible next proof steps in-between user interactions. It consists of two layers of blackboards with individual concurrent knowledge sources. The lower layer searches for instantiations of inference rule parameters within the actual proof state; the upper layer exploits this information to assemble a set of applicable inference rules and presents them to the user. The architecture also has mechanisms to adapt its behavior with respect to the current proof context and the availability of system resources. It furthermore allows for the integration of various automated components such as automated theorem provers, model generators or computer algebra systems. Moreover, the inference rule application can be automated itself, converting Ω-Ants into an automatic resource-adaptive reasoning system.

We also describe the integration of the Ω-Ants mechanism into the multi-strategy proof planner Multi to support traditional proof planning. In particular, we present how Ω-Ants can be employed to support interactive proof planning and to allow a search for applicable theorems from a mathematical knowledge base in parallel to the automatic proof planning process. Additionally, we present a means for soundly integrating certain symbolic computations into proof planning. The Ω-Ants architecture as well as all discussed combinations of reasoning techniques are implemented in the Ωmega theorem proving environment and for each combination we have carried out extensive case studies in group theory and finite algebra.

Kurzzusammenfassung

Die vorliegende Arbeit präsentiert einen neuen Ansatz zur Kombination verschiedener Beweistechniken, insbesondere von interaktivem und automatischem Theorembeweisen, sowie Beweisplanung und Computeralgebra, und deren Anwendung zur Beweisplanung in endlicher Algebra und Gruppentheorie. Die zentrale Struktur, mit deren Hilfe die Kombination durchgeführt wird, ist die hierarchische Blackboardarchitektur Ω-ANTS. Die Architektur verfügt über Mechanismen, sich sowohl bezüglich dem aktuellen Beweiskontext als auch der Ressourcenlage im System anzupassen. Darüberhinaus ermöglicht es eine Integration verschiedener automatischer Komponenten wie zum Beispiel automatischer Beweiser, Modellgenerierern oder Computeralgebrasystemen. Außerdem kann Ω-ANTS selbst als automatisches, ressourcenadaptives Beweissystem eingesetzt werden.

Weiterhin beschreibt die Arbeit die Integration des Ω-ANTS Mechanismus in den Multistrategiebeweisplaner MULTI zur Unterstützung von traditionellem Beweisplanen. Insbesondere kann Ω-ANTS zur interaktiven Beweisplanung benutzt werden und auch als Mechanismus, um während des automatischen Beweisplanens möglicherweise anwendbare mathematische Sätze in einer Wissensbasis zu suchen. Zusätzlich wird eine Methodik beschrieben, mit deren Hilfe symbolisches Rechnen auf korrekte, wenn auch eingeschränkte Weise in die Beweisplanung integriert werden kann. Sowohl die Ω-ANTS Architektur als auch die weiteren diskutierten Kombinationen von Beweistechniken wurden in der Beweisentwicklungsumgebung ΩMEGA implementiert und für jede der vorgestellten Kombinationen wurde eine ausführliche Fallstudie in der Gruppentheorie und endlichen Algebra durchgeführt.

Acknowledgments

Foremost, I want to thank *Jörg Siekmann*, who accepted me as a PhD student in the ΩMEGA group in Saarbrücken. This not only gave me the opportunity to pursue my research at a very renowned University, but the ΩMEGA group also provided a very stimulating environment that contributed to the success of my work.

I am also grateful to *Jacques Calmet* for accepting to be the second referee of this thesis and for his valuable comments on my work.

My deep gratitude is also to the late *Woody Bledsoe* whose enthusiastic lectures at the University of Texas at Austin raised my interest in Automated Theorem Proving in the first place.

For many fruitful discussions and collaborations in the ΩMEGA group I want to thank *Christoph Benzmüller*, *Lassaad Cheikhrouhou*, *Armin Fiedler*, *Helmut Horacek*, *Michael Kohlhase*, *Andreas Meier*, *Erica Melis*, and *Martin Pollet*. During the time when all the subgroups of the AG Siekmann where still located in the same building the cross-fertilization of ideas from different areas of Artificial Intelligence research significantly influenced my work. My thanks is especially but not exclusively to the following colleagues of the AG Siekmann outside the ΩMEGA group: *Serge Autexier*, *Dieter Hutter*, *Christoph Jung*, *Jürgen Lind*, *Heiko Mantel*, and *Axel Schairer*. For technical support and help with the implementation I am also indebted to *Heiko Fisch*, *Andreas Franke*, *Malte Hübner*, *Frank Theiß*, and *Jürgen Zimmer*. For carefully proof reading various parts and drafts of my thesis I thank *Serge Autexier*, *Christoph Benzmüller*, *Andreas Meier*, and *Martin Pollet*.

Furthermore, I owe a debt of gratitude to *Alan Bundy*, *Manfred Kerber*, and *Dominique Pastre* for enabling me to visit and work with their respective research groups. *Manfred Kerber* also provided me a one month refuge in Birmingham to complete my thesis.

I am grateful to the *Studienstiftung des Deutschen Volkes* for their support during my PhD studies. The Studienstiftung especially funded one year of PhD studies entirely as well as providing funds to visit *Alan Bundy's* group in Edinburgh. I also want to thank the *Deutscher Akademischer Austauschdienst* for funding my stay at *Dominique Pastre's* group in Paris.

Above all, I want to thank my wife *Catherine Vincent* for her support and patience during the whole time of my thesis and especially for taking more care of the kids than I could and for putting up with all my moods during the final months of my work. I have to apologize to *Elise* and *Clara* for not spending more time with them and in particular to *Elise* for using my computer for typing instead of for playing 'Bananas in Pyjamas' with her. Finally, I am deeply grateful to my parents for their support over the years.

Since I did some of the work reported here in collaboration with colleagues I chose to write 'We' throughout the thesis.

Summary

Motivation In the field of automated theorem proving and its applications in certain domains of mathematics many systems and techniques have been developed over the last decades. Automated theorem provers, interactive proof development environments, and proof planners have all had some success at least in restricted domains. However, up to now there has not existed a single system or approach that has reached such a degree of power and reliability that it can be seen as the *ultima ratio* for mechanized reasoning in mathematics. Therefore, there have been many attempts in the last decade to combine homogeneous and heterogeneous theorem provers, to integrate decision procedures and computer algebra into theorem proving and to parallelize theorem proving procedures. This thesis presents a novel approach to combining different reasoning techniques, in particular interactive and automated theorem proving, proof planning, and computer algebra systems and their application to proof planning in finite algebra and group theory.

Architecture The central structure to achieve the combination of different reasoning techniques is the hierarchical blackboard architecture Ω-Ants. This architecture was originally developed to support a user in interactive theorem proving by computing and suggesting the possible next proof steps in-between two user interactions. Ω-Ants consists a two layers of blackboards, each consisting of several individual, concurrent knowledge sources called agents. The task of the knowledge sources on the lower layer is to look for possible instantiations of parameters of proof rules. The knowledge sources on the upper layer use this gathered information in order to assemble a set of proof rules that may be applicable in the next proof step. The computed set of proof rules is then presented to the user who can choose one of the rules for application. The Ω-Ants architecture can adapt itself with respect to the given proof context as well as to the resources available in the overall system. This resource-adaptive behavior enables the integration of various automatic components such as automated theorem provers, model generators, or computer algebra systems. Furthermore, the application of the suggested proof rules itself can be automated, which turns Ω-Ants into an automatic, resource-adaptive proof planner.

Besides the application of Ω-Ants as an independent automated theorem prover and as a support tool for interactive theorem proving, the Ω-Ants mechanism can be an aid for traditional proof planning as well. This allows some aspects of the usual sequential approach of a proof planner to be enhanced using parallelism and concurrency. In this thesis we investigate two such aspects in the integration of Ω-Ants with Ωmega's multi-strategy proof planner Multi: (1) The use of the suggestion mechanism for interactive proof planning, and (2) the application of Ω-Ants as a mechanism to retrieve applicable mathematical theorems during automatic proof planning.

When supporting interactive proof planning, Ω-Ants is defined as a search algorithm for Multi and can then be parameterized with appropriate planning strategies. The applicability of single methods is then checked by Ω-Ants-agents and applicable methods are suggested to the user in a similar way as the regular proof rules are suggested during traditional interactive theorem proving.

By using Ω-Ants to retrieve applicable mathematical assertions during automatic proof planning we make particular use of the possibilities offered by the concurrency of the mechanism. This frees the actual, sequential proof planning algorithm from computationally expensive test of applicability for single theorems. The theorems of the knowledge base are automatically divided into different classes of theorems where each is assigned to a blackboard. The single theorems of the respective classes are checked for applicability in parallel, possibly using different criteria to decide applicability in a given proof context. Applicable theorems are gathered on the blackboards and suggested to the proof planner, which in turn exploits this information during the proof planning process.

In addition to the combination of proof planning with Ω-Ants we also present a method for a correct, albeit limited, integration of symbolic computation into proof planning. It is based on the idea of separating computation and verification and can thereby exploit the fact that many elaborate symbolic computations are trivial to verify. In proof planning the separation is realized by using a powerful computer algebra system during the planning process to do non-trivial symbolic computations. Results of these computations are checked during the refinement of a proof plan to a calculus level proof using a small, self-tailored system that gives us detailed interim information on its calculation. This information can be easily expanded into a checkable low-level calculus proof ensuring the correctness of the computation.

Case Studies For the evaluation of the architectural aspects and system integrations we we present four extensive case studies in the domain of finite algebra and group theory:

1. We demonstrate the use of Ω-Ants as an independent automatic proof planner with the proofs of several "equivalence" and "uniqueness" theorems from group theory. Thereby Ω-Ants uses a goal-directed search strategy for the higher order natural deduction calculus to decompose complex conjectures into smaller chunks, that can easily be solved by one of the integrated automated theorem provers.

2. The interactive proof planning combination of Multi and Ω-Ants is demonstrated with proofs of homomorphism theorems from group theory. The necessary planning methods are implemented in such a way that user interaction is limited to only the mathematically interesting steps.

3. A case study concerned with the automatic classification of residue class sets with respect to their algebraic properties demonstrates the integration of computer algebra into proof planning as well as the application of Ω-Ants to retrieve applicable theorems. The single combination aspects are demonstrated with several planning strategies, which implement different proof techniques. In detail we present three different strategies, namely exhaustive case analysis, equational reasoning and the application of given theorems.

4. Another case study illustrates the full strength of multi-strategy proof planning using both computer algebra systems and Ω-Ants, namely the interleaving of different planning strategies. This case study is concerned with isomorphism proofs and uses the results of the preceding case study. The

residue class sets already classified with respect to their algebraic properties are automatically classified into sets of isomorphic structures. The theorems that have to be proved during this classification process are of the form that two given residue class sets are either isomorphic or non-isomorphic to each other.

All the case studies are implemented in the ΩMEGA system and show the effectiveness of the Ω-ANTS architecture and the combination of reasoning techniques.

Zusammenfassung

Motivation Im automatischen Beweisens und insbesondere zu seiner Anwendung in Teilgebieten der Mathematik wurden in den letzten Jahrzehnten viele verschiedene Systeme und Techniken entwickelt. Automatische Beweiser, interaktive Beweisentwicklungsumgebungen sowie Beweisplaner hatten durchaus einen gewissen Erfolg zumindest in eingeschränkten Teilgebieten. Allerdings existiert für kaum ein mathematisch interessantes Gebiet ein einzelnes System, das dieses Gebiet vollständig abdeckt, indem es alle möglichen Probleme dieses Gebiets lösen kann. Daher gab es im Laufe des letzten Jahrzehnts eine Vielzahl verschiedener Ansätze zur Kombination verschiedener homogener oder heterogener Theorembeweiser, zur Integration von Entscheidungsprozeduren und Computeralgebra und zum Parallelisieren von Theorembeweisern. Die vorliegende Arbeit präsentiert einen neuen Ansatz zur Kombination verschiedener Beweistechniken, insbesondere von interaktivem und automatischem Theorembeweisen, sowie Beweisplanung und Computeralgebra, und deren Anwendung zur Beweisplanung in endlicher Algebra und Gruppentheorie.

Architektur Die zentrale Struktur mit deren Hilfe die Kombination durchgeführt wird ist die hierarchische Blackboardarchitektur Ω-Ants. Diese Architektur wurde ursprünglich zur Benutzerunterstützung im interaktiven Beweisen entwickelt, um während zweier Benutzerinteraktionen den nächsten möglichen Beweisschritt zu berechnen und vorzuschlagen. Ω-Ants selbst besteht aus zwei Ebenen von Blackboards, die jeweils mit einzelnen, nebenläufigen Wissensquellen – genannt Agenten – arbeiten. Dabei ist die Aufgabe der Wissensquellen der unteren Ebene in einem partiellen Beweis nach möglichen Instantiierungen von Parametern der einzelnen Beweisregeln zu suchen. Die Wissensquellen der obereren Ebene benutzen die so zusammengetragene Information, um die Menge der, im nächsten Beweisschritt anwendbaren Beweisregeln zusammenzustellen und diese dem Benutzer des Systems vorzuschlagen. Die Ω-Ants Architektur verfügt über Mechanismen, um sich sowohl bezüglich des aktuellen Beweiskontexts als auch der Ressourcenlage im System anzupassen. Dieses ressourcenadaptive Verhalten des Mechanismus ermöglicht auch eine kontrollierte Integration verschiedener automatischer Komponenten wie zum Beispiel automatischer Beweiser, Modellgenerierern oder Computeralgebrasystemen. Darüberhinaus kann die Anwendung der einzelnen, vorgeschlagenen Beweisregeln selbst automatisiert werden, was aus Ω-Ants ein automatisches, ressourcenadaptives Beweissystem macht.

Neben der Anwendung von Ω-Ants als eigenständigen automatischen Beweiser und zur Unterstützung des interaktiven Beweisens, kann der Ω-Ants Mechanismus auch als Hilfskomponente für die traditionelle Beweisplanung innerhalb von Ωmegas Multistrategiebeweisplaner Multi verwendet werden. Dabei kann Ω-Ants dazu benutzt werden, die herkömmliche, sequentielle Vorgehensweise eines Beweisplaners durch parallele Aspekte anzureichern. In der vorliegenden Arbeit wurden zwei dieser Aspekte näher betrachtet: zum einen die Benutzung von Ω-Ants zur interaktiven

Beweisplanung und zum anderen der Einsatz des Mechanismus zum Vorschlagen anwendbarer Theoreme im Rahmen der automatischen Beweisplanung.

Bei der Unterstützung der interaktiven Beweisplanung wird Ω-ANTS als ein Suchalgorithmus von MULTI definiert, der mit entsprechenden Planungsstrategien parametrisiert werden kann. Die Anwendbarkeit einzelner Methoden wird dann durch Ω-ANTS-Agenten geprüft und anwendbare Methoden werden dem Benutzer ähnlich wie die Beweisregeln beim herkömmlichen interaktiven Beweisen zur Auswahl vorgeschlagen.

Beim Einsatz von Ω-ANTS zur Suche in einem Beweiskontext anwendbarer mathematischer Sätze während der automatischen Beweisplanung wird besonders die Nebenläufigkeit des Mechanismus ausgenutzt, um den eigentlichen Planungsalgorithmus von den berechnungsintensiven Anwendbarkeitstests für die Sätze freizuhalten. Die Theoreme einer Wissensbasis werden automatisch in verschiedene Klassen eingeteilt, die jeweils einem Blackboard und seinen Wissensquellen zugeordnet werden. Die einzelnen Theoreme der verschiedenen Klassen werden dann nebenläufig mit möglicherweise verschiedenen Kriterien auf ihre Anwendbarkeit im aktuellen Beweiskontext hin überprüft. Anwendbare Theoreme werden auf den Blackboards gesammelt und dem Planer vorgeschlagen, der diese dann während der Beweisplanung berücksichtigen kann.

Zusätzlich zu den Kombinationen der Beweisplanung mit Ω-ANTS wird in der Arbeit eine Methodik beschrieben, mit deren Hilfe symbolisches Rechnen auf korrekte, wenn auch eingeschränkte Weise in die Beweisplanung integriert werden kann. Die Methodik basiert auf der Idee, daß bestimmte, komplexe symbolische Berechnungen relativ einfach überprüft werden können. Diese Idee kann man sich in der Beweisplanung zunutze machen, indem man die Methoden so konstruiert, daß während der Planungsphase komplexe algebraische Manipulationen mithilfe eines leistungsfähigen Computeralgebrasystems durchgeführt werden. Die Verifikation einer komplexen Berechnung geschieht dann innerhalb der Expansionsphase des Beweisplanes. Dabei wird der konkrete Berechnungsschritt in der einfacheren Gegenrichtung dadurch überprüft, daß er mittels des speziellen prototypischen Computeralgebrasystems $\mu\mathcal{CAS}$ nochmals berechnet wird. $\mu\mathcal{CAS}$ hat als Eigenschaft, daß es zu den Berechnungen zusätzliche Protokollinformationen ausgibt, die in ΩMEGA zur automatischen Verifikation der Berechnung benutzt werden können.

Fallstudien Zur Evakuierung der in der Arbeit vorgestellten Architekturaspekte werden vier ausführliche Fallstudien im Bereiche der endlichen Algebra und Gruppentheorie durchgeführt:

1. Die Benutzung von Ω-ANTS als eigenständiger automatischer Beweiser wird anhand der Beweise einiger Äquivalenz- und Eindeutigkeitstheoreme aus der Gruppentheorie demonstriert. Dabei werden mit einer zielgerichtete Suchstrategie für den Kalkül des natürlichen Schließens einer Logik höherer Stufe komplexe Theoreme zerlegt und die entstandenen einfacheren Teilprobleme von integrierten externen automatischen Beweisern gezeigt.

2. Die interaktive Beweisplanung mittels MULTI und Ω-ANTS wird mit dem Beweisen von Homomorphietheoremen aus der Gruppentheorie gezeigt. Dabei sind die verwendeten Planungsmethoden so implementiert, daß die Benutzerinteraktion möglichst nur für die mathematisch tatsächlich interessanten Schritte vonnöten ist.

3. Anhand einer Fallstudie zur automatischen algebraischen Klassifikation von Restklassenmengen mittels MULTI werden die Integration von Computeralge-

bra in die Beweisplanung und die Benutzung von Ω-Ants zur Theoremanwendung demonstriert. Die einzelnen Techniken werden innerhalb verschiedener Planungsstrategien implementiert, die jeweils unterschiedliche mathematische Beweistechniken realisieren. Dabei werden im einzelnen drei Strategien entworfen: vollständige Fallunterscheidung, Gleichheitsbeweisen und die Anwendung gegebener Theoreme.

4. In einer weiteren Fallstudie wird die volle Mächtigkeit der Multistrategie-Beweisplanung unter Verwendung von Computeralgebra und Ω-Ants gezeigt, indem Isomorphiebeweise mithilfe verschachtelter Anwendung der Strategien aus der vorangegangenen Fallstudie geführt werden. In der Fallstudie werden die Ergebnisse der Klassifikation der Restklassenmengen benutzt und Restklassen von gleicher algebraischer Struktur automatisch in Isomorphieklassen eingeteilt. Die dabei anfallenden Theoreme sind von der Art, daß zwei gegebene Restklassenmengen entweder zueinander isomorph oder nicht-isomorph sind.

Alle Fallstudien sind im Ωmega-System implementiert und zeigen die Effektivität von Ω-Ants und der Kombination verschiedener Beweistechniken.

Part I

Preliminaries

Chapter 1

Introduction

The history of science is characterized by an increasing degree of specialization into particular fields. LEIBNIZ is acknowledged as one of the last universal scholars, in the sense that he was aware and knowledgeable of all developments in all scientific fields of his time, whereas todays scientists became more and more specialized in their particular fields as the amount of knowledge grew. Specialization was the only way out also in the field of mathematics over the last two centuries: At the beginning of the nineteenth century mathematics was still a compact field, but since then it developed into the diverse field of the many sub-disciplines it is now. The same can be observed in the field of mechanized reasoning, which was influenced both by the diversity of modern mathematics and developments in the field as such.

Mechanized reasoning can be seen as an attempt to realize Leibniz's dream of a *'calculus ratiocinator'*, a universal language with the purpose to formalize and solve arbitrary reasoning processes. The evolution of modern logic starting in the nineteenth century with the work of BOOLE [40] and FREGE [89] and continued in the early twentieth century (cf. [212, 98, 109, 95]). The advent of the early computers at the time of the second world war and the logical developments particularly in proof theory marked the birth of the first inference machines, which were among the earliest existing artificial intelligence systems [159] to be presented at the Dartmouth Conference in 1956. One natural application domain of these systems was proving mathematical theorems.

Theorem proving systems for mathematics have since then been developed in a large variety. On the one hand this was influenced by the diversity of mathematics itself, which led to many special purpose systems that implemented different proof techniques suitable for particular mathematical domains. On the other hand the automated reasoning research itself has spawned various different branches leading to a strong diversification of the field. At the risk of oversimplifying we can, however, identify two major, albeit generally divergent, goals of research in mechanized reasoning. The first is the development of machine-oriented calculi that enable the construction of completely autonomous theorem provers. The second goal is the modeling of human problem-solving behavior and its cognitive aspects on a machine in order to build interactive proof checkers or plan-based automatic theorem provers. However, none of these approaches alone has so far reached a degree of power and reliability that it can be seen as the *ultima ratio* for mechanized reasoning in mathematics. There have been serious attempts to integrate human-oriented and machine-oriented reasoning by combining multiple proof techniques, and by enriching theorem provers with decision procedures and symbolic computation.

This thesis presents an approach to flexibly combine multiple reasoning techniques, such as automated and interactive theorem proving, proof planning, and symbolic computation, whose essence is a hierarchical blackboard architecture. The practicability of the approach is demonstrated by several case studies in the domain of finite algebra and group theory.

1.1 Machine-oriented Reasoning

Machine-oriented theorem provers are essentially automatic provers based upon some computer-oriented inference system such as the *resolution* principle [181]. The most important aspect of the resolution calculus is that it replaces nondeterministic instantiation of variables by goal directed, algorithmic unification. Other machine-oriented calculi are, for instance, *tableaux* [192] or *connection methods* [130, 188]. Modern systems derive their strength from their ability to maintain and manipulate very large search spaces based on sophisticated indexing techniques [103, 176]. Their strength can be truly remarkable, but their performance in real mathematics is still, after more than forty years of research and steady improvements, rather weak.

Today there are many general purpose theorem provers for different logics, for example for propositional logic there are the SAT-based provers that use the *Davis-Putnam procedure* [69] to compute satisfiability of a given propositional formula. Two prominent representatives of this class are, for instance, SATO [217] and MACE [140]. For first order logic a myriad of systems have been developed such as MKRP [171], OTTER [143], BLIKSEM [70], or SPASS [209] based on the resolution calculus, which was enhanced by *paramodulation* [180], *superposition* [16], and connection graphs [130, 188]. Systems based on other calculi are, for example, SETHEO [184], which uses a tableau calculus, or LEANCOP [164], which uses the connection method. Similarly for higher order logics there are systems based on the appropriately adapted resolution principle [3, 25] such as the LEO system [26] or on the connection method [8] such as TPS [9].

Besides general purpose theorem provers there were also systems built for special proof techniques and application domains. For example, there is a whole subfield of automated theorem proving concerned with equational theorem proving. Here term rewriting systems have been developed with the purpose to transform a set of equations with procedures such as *Knuth-Bendix completion* [128] into a system that guarantees the existence of unique normal forms. This way any term can be rewritten into a unique normal form and thus shown whether an equality holds or not. Prominent representatives for term rewriting systems are, for instance, WALDMEISTER [110], ELAN [41], and EQP [143]. Other typical domains for special purpose provers are, for instance, purely inductive theorem proving and geometry theorem proving. Examples for inductive theorem provers are NQTHM [44], ACL2 [121], or INKA [13], which are particularly useful for software verification. Instances of geometry theorem proving systems are GEOMETRY EXPERT [61] and GOETHER [208], which use algebraic methods such as *Wu's method* [216] or *Gröbner bases* [133] – GOETHER is for instance just a package for the computer algebra system MAPLE – to prove theorems in elementary geometries.

Although automated theorem provers based on machine-oriented calculi have reached a respectable strength and had success in proving previously open problems (e.g., the Robbins conjecture was proved to be valid by EQP [143]), none of the existing systems shows the strength and reliability such that it can solve all problems in a certain domain.

1.2 Human-oriented Reasoning

The limits and unpredictability of automated proof search has led to the development of interactive theorem proving environments, which provide means for a user to interactively construct and check proofs in a logically sound calculus. One of the earliest interactive provers was the AUTOMATH system [206] developed by DE BRUIJN in the 1970s. More recent systems like NUPRL [65], ISABELLE [166], HOL [102], PVS [165], IMPS [81], and COQ [66], provide expressive, generally higher order, languages and more human-oriented calculi such as *natural deduction* [172] or *sequent calculi* [95]. The disadvantage of these systems is, however, that proofs have to be painstakingly derived on a very fine-grained level which can lead to many, often tedious interactions. Therefore, many interactive systems offer facilities to define so-called *tactics*, first used in LCF [101], which are programs that manipulate the current state of the proof by the application of a whole series of calculus rules. In this way a single user interaction, namely the call of a tactic, results in a sequence of proof steps. Tactics can be used to encode logically recurring proof patterns but also to a limited extend to incorporate human-oriented proof techniques. However, the proof in interactive proof development systems is essentially provided by the user with relatively little help from the machine.

In order to remedy this situation BUNDY developed the notion of *proof planning* [52]. His idea is essentially to have *methods*, which are tactics enriched with pre- and postconditions that specify the applicability of the tactic as well as its effects on the proof state. Theorems are then proved by automatically constructing appropriate combinations of methods, so-called proof plans, using artificial intelligence planning techniques. There are basically two directions in proof planning, one is to simply automate traditional tactical theorem proving by deriving methods as general and as broadly applicable as possible. The other is to model human techniques with methods by incorporating domain-specific mathematical knowledge. The former approach is implemented in CLAM [54] and λCLAM [179] whereas the latter is the paradigm of the ΩMEGA system [22]. The abstract calculus implemented by methods is generally not complete and, moreover, since methods can be under-specified, constructed proof plans are not necessarily correct. Therefore, proof plans have to be executed in order to construct machine-checkable proofs in an underlying sound calculus.

1.3 Integration of Reasoning Techniques

As it turns out neither pure machine-oriented automated theorem proving nor human-oriented interactive and plan-based reasoning are powerful enough to be seen as the *ultima ratio* for mechanized reasoning. In fact, a combination of already developed theorem proving techniques as well as their enrichment with other reasoning techniques such as symbolic computation or constraint solving is more desirable. Consequently, over the last decade there have been many attempts at implementing such combinations either in newly developed systems or by integrating already existing systems and by providing environments that facilitate their integration.

1.3.1 Integration of Deduction Systems

One method to enhance the power and acceptance of interactive theorem provers is to integrate automated theorem proving in order to discharge appropriate subgoals during large proofs. This can be achieved essentially in two ways: One is to design powerful tactics that implement automated proof procedures. Tactics such as *blast* in the HOL [102] system or *grind* in PVS [165] incorporate nearly full scale first order automated theorem proving into the respective system. The calculus of TPS [9] contains an inference rule named *RuleP* that offers a way to automatically discharge predicate logic subgoals.

Another way to partially automate the interactive theorem proving process is to integrate already existing systems into the interactive environments. This has the advantage that one can reuse existing state of the art technology as well as to integrate several different means of automation. For instance, the ILF system [67, 68] integrates several different first order theorem provers, the ΩMEGA system [22] integrates a variety of provers for first and higher order logic as well as model generators, and the PROSPER environment [72] offers provers for first order logic and inductive reasoning as well as model generators.

Besides the integration of automated theorem proving into interactive environments there are also approaches to integrate environments for their mutual benefits. Examples of such integrations are the interface between HOL and NUPRL [82] that enables the exchange of proofs in classical and constructive logic, the integration of HOL and CLAM [191] that allows CLAM proof plans to be translated into HOL tactics, or the interface between ΩMEGA and TPS [21] that enables the exchange of concepts in the respective knowledge bases.

1.3.2 Integration of Deduction and Computer Algebra

There is research and development with the aim to integrate other reasoning techniques such as decision procedures or constraint solving into theorem proving. Compare, for instance, the integration of *Preßburger arithmetic* into the NQTHM inductive theorem prover [46]. During the last ten years there has been also an interest in the integration of deduction systems (DS) and computer algebra systems (CAS).

There are two intentions for the integration of deduction and computer algebra: One is to provide some guarantee for correct computations by enriching CAS with deduction (i.e., DS $\subseteq$ CAS) or to enhance the computational power of DS (i.e., CAS $\subseteq$ DS), which are notoriously weak in that respect. Although experiments for both have been carried out, we are mainly interested in the latter scenario (for some examples of the former see [2, 170, 204, 10]). For the integration of symbolic computation into DS there exist basically three approaches: (1) To fully trust the CAS, (2) to use the CAS as an oracle and to try to reconstruct the proof in the DS with purely logical inferences, and (3) to extend the DS with a symbolic computation component that either can be fully trusted or that produces output that can be checked for its correctness.

In the first category (c.f. [196, 63, 17, 18, 48]) one essentially trusts that the CAS works properly, hence their results are directly incorporated into the proof. All experiments in this category are at least partly motivated by achieving a broader applicability range of automated reasoning and this objective has been definitively achieved, since the range of mathematical theorems that can be formally proved by the combined systems is much greater than by the DS alone. However, CAS are very complex programs and therefore trustworthy only to a limited extent, so that the correctness of proofs in such a hybrid system can be questioned, particularly as it is often difficult to check all the side conditions and constraints, such as not dividing by zero, etc. This is not only a minor technical problem, but will remain unsolved for the foreseeable future since the complexity (not only the code complexity, but also the mathematical complexity) does not permit a verification of the program itself with currently available program verification methods.

The second category [105, 107] is more conscious with respect to the role of proofs, and uses the CAS only as an oracle, with the result, that the correctness can then be checked deductively. While this certainly solves the correctness problem, this approach has only a limited coverage, since even checking the correctness of a calculation may be out of scope for most DS without additional information. Indeed from the point of applicability, the results of the CAS help only in cases, where the verification of a result has a lower complexity than its discovery, such as prime factorizations, solving equations, or indefinite symbolic integration. In an alternative approach that formally respects correctness, but essentially trusts CAS, an additional assumption standing for the respective CAS is introduced, so that essentially formulae are derived that are shown modulo the correctness of the computer algebra system at hand (e.g., see [106]).

A third approach of integrating CAS into DS, consists in the meta-theoretic extension of the reasoning system as proposed, for instance, in [45, 114] and realized in NUPRL. In this approach a constructive mechanized reasoning system is basically used as its own meta-system and the constructive features are exploited to construct a correct computer algebra system. Bridge rules between the ground and the meta-system are employed to integrate the CAS thus constructed. The theoretical properties of the meta-theoretic extension guarantee that if the original system was correct then the extended system is correct too. Similar is some work done in the COQ system where algebraic algorithms are formally specified and certified in a constructive logic and the specifications are compiled into executable, correct code. Experiments include a certified version of *Buchberger's algorithm* [202] as well as the formal development of basic polynomial algorithms [42, 43]. In the same category we can also see the approach presented in [123, 124] where a self-tailored CAS is implemented that generates intermediate output during its computation. This output can then be translated into tactics of the DS and expanded to a calculus level proof, which in turn can be machine-checked. A disadvantage compared to the other two approaches is that it is not possible to employ any of the existing CAS, but it is necessary to (re)implement it either in the formal system of the basic DS or with an appropriate enriched output for the interim information.

In this thesis we present a pragmatic approach at integrating CAS into DS, more precisely into proof planning, that combines aspects of the approaches (2) and (3). It is based on the assumption that computation and verification can be separated and this approach can thus exploit the fact that many elaborate symbolic computations are trivially checked. In proof planning the separation is realized by using a powerful, existing CAS during the planning process to do non-trivial symbolic computations. Results of these computations are checked during the refinement of a proof plan into a calculus level proof using a small, especially implemented system that gives us interim information on its calculation. This information can be easily expanded into a checkable low-level calculus proof ensuring the correctness of the computation.

1.3.3 Frameworks for Integration

The need to combine different DS in a single environment that is flexible enough to handle both replacement and addition of systems has led to the concept of **O**pen **M**echanized **R**easoning **S**ystems [97]. In OMRS, theorem provers can be viewed as replaceable plug and play components. It turned out that theorem proving systems for a plug and play environment have to be separated into distinct components for control and logic or computation. Thus, it is practically impossible to integrate any monolithic system without redesigning major parts. Moreover, commercial systems where the sources are not available cannot be re-engineered and are therefore lost for an integration.

A framework for establishing the semantics of intimately integrated deduction and computation systems was presented by Homann and Calmet in [111, 113] and a classification of logical and symbolic computation systems as well as the aspects of their communication and cooperation has been developed in [56]. Homann and Calmet also generalized the concept of OMRS first to an open environment for doing mathematics [112] and together with Bertoli and Giunchiglia to the concept of **O**pen **M**echanized **S**ymbolic **C**omputation **S**ystems [32, 33]. In OMSCS computer algebra systems, theorem provers, and their integration can be soundly expressed, which has been demonstrated with a case study integrating the deduction system Isabelle and the general purpose computer algebra system Maple.

While this work develops a *semantics* to support the integration of different systems, there are also frameworks that solely provide the infrastructure to integrate existing systems. For instance, in the MathWeb architecture [88] systems can be integrated as so called mathematical services by encapsulating them into agent shells. MathWeb then manages the communication between the systems by providing a uniform KQML-like language [85] but has no specific requirements for the actual content of the communication. Thus, virtually anything can be communicated between the systems. Although MathWeb takes care of the distribution of the services over the Internet it does not automatically enable the cooperation between systems, instead coordination has to be provided by a requesting system. For instance, a system such as Ωmega can send requests to selected mathematical services via MathWeb thereby treating these services as slaves. MathWeb also exhibits limited abilities for resource management of services, however, this is restricted to a static time out for requests. Similar to MathWeb are the Mathematical Softwarebus [57] from Homann and Calmet and the Logic Broker architecture [11] from Armando and Zini. The advantage of the latter is that it uses the more established Corba [187] protocol for distribution.

The approach we shall present in this thesis differs from the above in two points: Firstly both the cooperation and competition of reasoning techniques and of integrated reasoners is automatic; that is, computation by the integrated components are implicitly triggered by the state of the problem at hand and not explicitly by a user. Secondly, our approach incorporates already existing systems without the need for re-engineering. However, in order to derive a machine-checkable proof object we have to rely on mechanisms in order to translate any reasoning step provided by some external reasoner into the uniform representation of Ωmega's natural deduction calculus. For the integration and distribution of external systems we exploit the facilities provided by Ωmega's MathWeb architecture and therefore have the integrated components run in parallel.

1.4 Parallel Theorem Proving

The field of parallel theorem proving has drawn a growing attention over the last decade. Several calculi have been extended to support parallelism and tested in practice. For instance, a parallel version of the Davis-Putnam procedure has been developed for the PSato system [218] or a parallel Knuth-Bendix algorithm has been implemented in PaReDuX [50, 51]. In other approaches the same theorem provers have been run in parallel with different settings simulating different theorem proving strategies. For example P-SeTheo is a prover that parallelizes the tableau-based theorem prover SeTheo [213]. For a more profound overview on parallel, mainly first order theorem proving systems as well as a taxonomy of parallelism in theorem proving we refer the reader to [36, 37]. Here we shall only discuss some of the heterogeneous approaches that are relevant for our work.

The Techs approach [74] realizes cooperation between a set of heterogeneous first order theorem provers. Partial results are exchanged between the different theorem provers in the form of clauses, and different referees filter the communication at the sender and receiver side. This system clearly demonstrated that the joint system is much stronger than the individual systems. Techs notion of heterogeneous systems, however, is restricted to a first order context only and neither higher order provers, computer algebra systems, nor model generators can be integrated. Predecessors of Techs are Teamwork [75] and Discount [14, 76], which are more machine-oriented and thus do not allow for interaction. Interaction and automation is addressed by the combination of Ilf and Techs as described in [73].

In [87], Fisher and Ireland propose an agent-based approach to proof planning that is motivated by a fine-grained parallelization of the proof planning process more than the distribution aspect. They propose a society of agents that are organized by a contract net architecture, building on earlier studies of Fisher [86] on agent-based theorem proving.

In most of these approaches the construction of a single proof object is given up and replaced by the more simple goal of solving the problem without reconstructing a proof. This is, however, not desirable for us, since we do not only want explicit user interaction but also to construct proofs that are both machine-checkable and presentable to a user. Moreover, the systems integrated into our architecture are very heterogeneous in the sense that we allow for theorem provers with various calculi and logics, computer algebra systems and model generators. Therefore, the information that is exchanged is on the level of subproblems that are maintained in a central proof object, and not at the very low level such as clauses.

Since our approach is geared towards proof construction and user interaction, we opted for a proof centered approach since this supports in particular user interaction and the construction of a uniform proof object. Therefore, we chose a blackboard architecture that allows a flexible distribution without loss of the desired centralization.

1.5 Blackboard Systems

Blackboard architectures have been developed in the nineteen-eighties as a simple yet powerful means to deal with uncertain data and to apply a non-deterministic solution strategy. In the blackboard model solutions are assembled by a variety of *knowledge sources*, which do not have to be of a uniform composure. Solutions are assembled on the blackboard by cooperation of the different knowledge sources, whose computations are scheduled on the board. Nearly all blackboard models have a hierarchical structure for both the solution space and the knowledge sources. This enables a propagation of better solutions in the hierarchy and also an anytime character in the sense that the more computations are performed by the knowledge sources the better the eventual solution becomes. (Compare [79] for an extensive introduction to blackboards.)

The first blackboard architectures were the HearsayII [80] or the Hasp [161] architectures, where the former was used for speech recognition and the latter for ocean surveillance with sonar sensors. Both consisted of a single blackboard and a set of hierarchical structured knowledge sources. Later blackboard architectures were then composed of several blackboards for different tasks. For instance, the BB1 architecture is composed of two blackboards, one for the scheduling tasks and one to assemble the solution.

The early blackboard systems had no concurrency for the computation in the knowledge sources. These ideas were first picked up for systems such as CAGE [160] and POLIGON [178], which were enriched by both cooperation and concurrency between the knowledge sources. In particular, the POLIGON architecture enables a very flexible integration of various blackboards and their knowledge sources and resembles more a modern multi-agent architecture than a classical blackboard approach.

The architecture we shall present in this thesis is influenced both by the hierarchical approach of the earlier blackboard architectures as well as by parallelism as in the POLIGON and CAGE architectures. It is composed of two layers of independent blackboards and parallel knowledge sources. Solutions are therefore assembled in a two stage process and propagated from the lower to the upper layer, where on the upper layer we always have the heuristically best solution that has been computed so far. Moreover, the use of several independent blackboards and parallel knowledge sources allows us to model both competition and cooperation of knowledge sources.

1.6 Theorem Proving in Group Theory and Finite Algebra

In the history of automated theorem proving many specialized provers have been built for different mathematical domains including group theory and finite algebra. Moreover, many non-specialized theorem proving systems have been successfully applied to prove theorems in group theory. A full overview of automated reasoning concerned with group theory is beyond the scope of this introduction, we rather give a short historical overview and a short, incomplete account of recent work in that area.

The first theorem proving systems specialized in group theory were developed in the 1960s. In [215] ROBINSON, WOS, and CARSON present an automated theorem prover that proved some theorems in group theory in a specialized abstract calculus. In the following year NORTON — a PhD student of MINSKY — developed the ADEPT-system [162], a first order logic theorem prover specialized on group theory. The specialized heuristics were essentially simple term rewriting steps corresponding to the application of equations like the group axioms. Other early attempts at theorem proving in group theory is, for instance, the work by ZHERLOV and MARTÝANOV [219].

Especially the proper treatment of equality within proofs is still a difficult problem for many automated systems [136]. This is one of the reasons why problems from group theory, albeit mathematically trivial , are still considered to be hard problems for automated theorem provers (cf. [138, 167]) and are often used as challenge problems for term rewriting systems (i.e., theorem provers that are specialized on equational reasoning). Compare, for instance, some of the problems in the TPTP [197, 198], a library of benchmark problems for first order theorem provers.

One important means to construct term rewriting systems is the already mentioned Knuth-Bendix algorithm [128] whose method is to direct and complete sets of equational axioms into a confluent and terminating term rewriting system. Special orderings on terms are used for termination, which vary depending, for instance, on particular mathematical domains. Work in this area takes equations from group theory as starting point. For instance, in [49] BUENDGEN describes an application of the Knuth-Bendix algorithm to finite group theory. In [90] FUCHS presents a method using Knuth-Bendix completion together with additional goal directed heuristics to prove theorems on lattice ordered groups. All this work is concerned with proofs of general theorems about groups. However, if one is concerned with theorems about particular instances of groups it is possible to construct special term rewriting systems depending on the particular group. Such a method is, for example, presented by MARTIN in [137], where a uniquely determined and convergent term rewriting system can be constructed for a group, given its generators.

Besides inference techniques for equational problems based on term rewriting there exist also special calculi that integrate equations directly into their rules. One such calculus is the *superposition* calculus by BACHMAIR, GANZINGER, LYNCH, and SNYDER [16], where equality treatment is integrated into a resolution calculus similar to paramodulation. Additionally there exists a strict order on terms that directs equations in a way such that all term rewriting steps terminate. Extensions of this calculus have been developed to cater for particular mathematical domains: For instance WALDMANN has developed a superposition calculus for monoids [93] and STUBER for groups [195].

There is also work in interactive theorem proving in the context of algebra and group theory. JACKSON has developed computational abstract algebra in the NUPRL proof development system [65], in particular the necessary concepts to develop the basics of polynomial algebra. SCHWARZWELLER [185] presents work on the formalization of basic abstract ring theory in the MIZAR library [205, 182], a mathematical knowledge base with an interactive proof checker, in order to verify generic algebraic algorithms, such as the generic Euclidean algorithm. The formal proof of *Sylow's first theorem* in the interactive theorem prover ISABELLE [166] is described in [119], which is based on KAMMÜLLER's formalization of group theory and prime number theory in ISABELLE. A formal derivation of the *Fundamental Theorem of Algebra* has been carried out by GEUVERS, WIEDIJK, and ZWANENBURG [96] in the COQ system.

The work discussed so far is mainly concerned with formalizing existing algebraic theory and constructing proofs for already known theorems. However, there is also more explorative work in the context of group theory and finite algebra. For instance, in the quest for minimal axiomatizations for algebraic entities, in particular groups, automated theorem provers have been employed to find single axioms, from which all necessary properties of a group can be derived [158, 132]. Especially a first order theorem prover was successfully applied by MCCUNE to find several of such axioms [139].

There is also work on exploration and automated discovery in finite algebra that is concerned with the discovery of particular algebraic structures that satisfy given properties. For instance, in [91, 141, 190, 218] model generation techniques are used to tackle quasi-group existence problems. In particular, systems such as FINDER [189] and SATO [217] were successfully employed to solve some open problems in quasi-group theory. MCCUNE and PADMANABHAN [144] give an account of the use of the first order automated theorem prover OTTER to assist the construction of non-associative algebras in every day mathematical practice. There is also work by GOMES, SELMAN, CRATO, and KAUTZ [99] where constraint solving techniques are used to complete quasi-group multiplication tables. The motivation for all this work is roughly to specify certain properties of an algebra and then try to automatically construct a structure that satisfies the required properties. Thus, the constructed algebra might actually be a new discovery.

1.7 Summary and Outline of the Thesis

This thesis is concerned with the combination of several reasoning techniques to be applied in group theory and finite algebra. In particular, we combine interactive and automated reasoning, proof planning and symbolic computation. Our means to achieve this combination is a hierarchical blackboard architecture called Ω-Ants. This architecture has been developed originally to support a user in interactive theorem proving to search for the next possible proof step. It consists of two layers of blackboards with individual concurrent knowledge sources. The lower layer searches for instantiations of command parameters within the actual proof state. The upper layer exploits this information to assemble a set of possibly applicable proof steps and presents them to the user. The architecture has also mechanisms to adapt its behavior with respect to the current proof context and the availability of system resources. It especially allows for the integration of various components such as automated theorem provers, model generators, or computer algebra systems. Moreover, the command application can be automated itself converting Ω-Ants into an automatic, resource-adaptive reasoning system.

We also describe the integration of the Ω-Ants mechanism into the multi-strategy proof planner Multi [155] to support traditional proof planning. In particular, we present how Ω-Ants can be employed firstly for interactive proof planning and secondly to seek applicable theorems from a mathematical knowledge base in parallel to the automatic proof planning process. Additionally, we present how certain symbolic computations can be soundly integrated into proof planning. The Ω-Ants architecture as well as all discussed combinations of reasoning techniques are implemented in the Ωmega theorem proving environment and for each combination we have carried out extensive case studies in group theory and finite algebra.

The thesis consists of three parts. The first part presents an introduction to the Ωmega system, the logic of the Ωmega system and the notions of proof planning to which we shall refer to throughout the rest of the thesis.

The second part of the thesis is concerned with architectures for the combination of reasoning techniques. It consists of two chapters: Chapter 3 introduces the Ω-Ants blackboard architecture, our means to combine interactive and automated reasoning. Chapter 4 presents how Ω-Ants can be used within the multi-strategy proof planner Multi to determine method applicability for interactive proof planning and to check for applicable theorems from the mathematical knowledge base in parallel to the automatic proof planning process. Furthermore, this chapter presents how non-trivial computations of regular computer algebra systems can be soundly integrated into proof planning.

In the last part of the thesis we present several case studies in the domain of group theory and finite algebra to illustrate different aspects of the combination. In detail, the third part contains four chapters, each describing a different case study: Chapter 5 is concerned with equivalence and uniqueness proofs of algebra that illustrate the use of Ω-Ants as an automated theorem prover. Chapter 6 elaborates how interactive proof planning can be performed with Ω-Ants using homomorphism theorems from group theory as a case study. Finally, the chapters 7 and 8 present a case study for the application of Ω-Ants for knowledge base queries during proof planning as well as the integration of symbolic computation into proof planning. Chapter 7 contains proofs for simple algebraic properties of residue classes using straightforward proof planning techniques, whereas chapter 8 presents isomorphism proofs between residue classes that employ the full power of multi-strategy proof planning.

Chapter 2

An Introduction to ΩMEGA

ΩMEGA [22] is a theorem proving environment for interactive as well as automated proof development. Its distributed architecture allows for the cooperation and integration of external systems such as a classical deduction system or a computer algebra system. The main purpose of ΩMEGA is to construct proof objects, which are machine-checkable for correctness. This is done in a GENTZEN-style natural deduction calculus [95] based on a variant of CHURCH's simply typed higher order lambda calculus [62]. Proof construction itself is performed at an abstract level where the user can employ a variety of tools, such as interactive tactical theorem proving, automatic proof planning or the application of external systems.

In this chapter we show ΩMEGA's logic, introducing its syntax, semantics, and a natural deduction calculus. We then describe ΩMEGA's proof objects and how they can be constructed. Also we give a brief overview of the concepts in ΩMEGA's knowledge base that are relevant for proof construction. Moreover, we explain the tactical theorem proving facilities and give a brief introduction to proof planning and ΩMEGA's particularities of knowledge based and multi-strategy proof planning. Finally, we explain how external reasoners can be employed during proof construction.

2.1 ΩMEGA's Logic

ΩMEGA's basic logic is a higher order logic based on a simply typed lambda calculus. Proofs are constructed in a GENTZEN-style natural deduction. We first define the syntax and semantics for the logic and then give the inference rules of the natural deduction calculus.

2.1.1 Syntax

Definition 2.1 (Types): Let $\mathcal{T}_B$ be a nonempty, finite set of symbols. We define the set $\mathcal{T}$ of *types* inductively as the smallest set containing $\mathcal{T}_B$ and all types of the form $\alpha \rightarrow \beta$, where $\alpha, \beta \in \mathcal{T}$.
We call the elements of $\mathcal{T}_B$ *base-types* and types of the form $\alpha \rightarrow \beta$ *functional types*. □

In the sequel we will always assume a fixed set of base-types $\mathcal{T}_B$ and types $\mathcal{T}$ with $\{o, \iota\} \subset \mathcal{T}_B$. Here o denotes the type of *truth-values* and ι denotes the type of *individuals*. However, $\mathcal{T}_B$ can be extended by other special types, for instance, in ΩMEGA there exists a special type ν denoting the type of numbers. We shall write small Greek letters for the syntactical variables denoting elements of $\mathcal{T}$.

Notation 2.2: A type of the form $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta$ is bracketed to the right and thus corresponds to $(\alpha_1 \to (\alpha_2 \to \ldots \to (\alpha_n \to \beta)\ldots))$. We may omit brackets and arrows altogether and write $\alpha_1\alpha_2\ldots\alpha_n\beta$, when no ambiguity is thereby introduced.

Definition 2.3 (Typed sets): A set of sets of symbols $\Gamma = \Gamma_{\mathcal{T}} := \{\Gamma_\alpha | \alpha \in \mathcal{T}\}$ is called a *typed collection of sets* over $\mathcal{T}$. We call Γ *disjoint* if we have $\Gamma_\alpha \cap \Gamma_\beta = \emptyset$, for $\alpha \neq \beta$ and $\alpha, \beta \in \mathcal{T}$.
The mapping $\tau: \Gamma \to \mathcal{T}$ is called a *type function* if for each $\alpha \in \mathcal{T}$ and each $f \in \Gamma_\alpha$ holds: $\tau(f) = \alpha$. Conversely, a type function $\tau : \mathcal{M} \to \mathcal{T}$ induces a disjoint typed collection $\mathcal{M}_{\mathcal{T}} = \{\mathcal{M}_\alpha\}$ for $\mathcal{M}_\alpha = \tau(\alpha)$.
Given two typed collections of sets $\mathcal{D}, \mathcal{E}$ over the same set of types $\mathcal{T}$, we call a collection of functions $\mathcal{I} := \{\mathcal{I}_\alpha : \mathcal{E}_\alpha \to \mathcal{D}_\alpha | \alpha \in \mathcal{T}\}$ a *typed function* $\mathcal{I} : \mathcal{E} \to \mathcal{D}$. □

We shall write an element $c \in \mathcal{D}_\alpha$ of a typed set $\mathcal{D}_\alpha$ as c_α in order to indicate that it is of type α. We will, however, convey the type information of a typed element only once or even omit it if its type is obvious from the context.

Definition 2.4 (Signature): Let Σ be a disjoint typed collection of countably infinite sets over $\mathcal{T}$ then Σ is called a *signature* over $\mathcal{T}$ and the elements of the Σ_α are called *constants*.
Σ contains in particular the *logical constants* $\{\neg_{oo}, \vee_{oo}, \Pi_{\alpha oo}, \imath_{\alpha o\alpha}\} \subseteq \Sigma$. □

The symbols $\neg$, $\vee$, and Π are called negation, disjunction and quantifier, respectively. $\imath$ is the description operator as introduced in [5]. Its purpose is to pick the unique element out of a singleton set. We shall axiomatize and explain this more detailed in section 2.1.3.

Note that the quantifier $\Pi_{\alpha oo}$ and the description operator $\imath_{\alpha o\alpha}$ in definition 2.4 depend on the type of their argument. Therefore, there exists for every type $\alpha \in \mathcal{T}$ exactly one quantifier Π^α and one description operator $\imath^\alpha$. We call such a definition where α is not fixed a *polymorphic definition*.

The preceding definitions allow us to regard the signature as a union of typed sets of constant symbols. Since they are disjoint we can uniquely determine the exact type of each constant with the type function τ. Moreover, the use of polymorphic definitions enables us in most cases to state the elements of Σ in a finite way although it is collection of countably infinite sets.

Definition 2.5 (Well-formed formulas): Let Σ be a signature over $\mathcal{T}$ and $\mathcal{V}$ a collection of typed sets over $\mathcal{T}$ with countable infinitely many elements. We call $\mathcal{V}$ the *set of typed variables*. For each type $\alpha \in \mathcal{T}$ we inductively define the set $\mathbf{wff}_\alpha(\Sigma)$ of *well-formed formulas* as

(i) $\Sigma_\alpha \subseteq \mathbf{wff}_\alpha(\Sigma)$,

(ii) $\mathcal{V}_\alpha \subseteq \mathbf{wff}_\alpha(\Sigma)$,

(iii) with $\mathbf{A}_{\alpha\to\beta} \in \mathbf{wff}_{\alpha\to\beta}(\Sigma)$ and $\mathbf{B}_\alpha \in \mathbf{wff}_\alpha(\Sigma)$ is $(\mathbf{AB}) \in \mathbf{wff}_\beta(\Sigma)$,

(iv) if $\mathbf{A}_\alpha \in \mathbf{wff}_\alpha(\Sigma)$ and $X \in \mathcal{V}_\beta$ then $(\lambda X_\bullet \mathbf{A}) \in \mathbf{wff}_{\beta\to\alpha}(\Sigma)$.

The set of all well-formed formulas over the signature Σ can be defined as $\mathbf{wff}(\Sigma) = \bigcup_{\alpha\in\mathcal{T}} \mathbf{wff}_\alpha(\Sigma)$. □

We call formulas of the form $(\mathbf{AB})$ *applications* and formulas of the form $(\lambda X_\blacksquare \mathbf{A})$ *λ-abstractions* or simply *abstractions*. The elements of $\mathbf{wff}_o(\Sigma)$ will be called *propositions*.

Notation 2.6: The square dot '$_\blacksquare$' in $(\lambda X_\blacksquare \mathbf{A})$ divides the *λ-bound* variable X from its *scope* $\mathbf{A}$. It corresponds to a left bracket whose mate is as far to the right as possible until a right bracket is reached whose mate is left of the λ-binder.

Notation 2.7: Until the end of this thesis we will use infix notation instead of prefix notation when it does not lead to ambiguities. For instance, we write $(\mathbf{A} \vee \mathbf{B})$ instead of $(\vee\mathbf{AB})$. Likewise, to ease readability we will omit brackets whenever possible and write function application in the more mathematical style of $f(c)$ instead of (fc).

Definition 2.8 (Free variables): Let $\mathbf{A}, \mathbf{B} \in \mathbf{wff}(\Sigma)$ and let $Z \in \mathcal{V}_\mathcal{T}$. The occurrence of a variable Z is called *bound* in $\mathbf{A}$ iff it is in a sub-formula of the form $\lambda Z_\blacksquare \mathbf{B}$ in $\mathbf{A}$. In case an occurrence of Z in $\mathbf{A}$ is not bound we call it *free* in $\mathbf{A}$. We define the set of all variables with free occurrences in $\mathbf{A}$ as the set of free variables of von $\mathbf{A}$, $\mathbf{FV}(\mathbf{A})$. □

Definition 2.9 (λ-conversions): Let $\mathbf{A} \in \mathbf{wff}_\alpha(\Sigma)$, $\mathbf{B} \in \mathbf{wff}_\beta(\Sigma)$ and let $X, Y \in \mathcal{V}_\beta$. For the formula $\mathbf{A}$ we define three rules of *λ-conversion*:

(i) $\lambda X_\blacksquare \mathbf{A} \rightarrow_\alpha \lambda Y_\blacksquare [Y/X]\mathbf{A}$, provided $Y \notin \mathbf{A}$ (*α-conversion*)

(ii) $(\lambda X_\blacksquare \mathbf{A})\mathbf{B} \rightarrow_\beta [\mathbf{B}/X]\mathbf{A}$ (*β-reduction*)

(iii) $(\lambda X_\blacksquare \mathbf{A}X) \rightarrow_\eta \mathbf{A}$, if $X \notin \mathbf{FV}(\mathbf{A})$ (*η-reduction*)

Here the notation $[\mathbf{B}/X]\mathbf{A}$ means that all free occurrences of the variable X in $\mathbf{A}$ are substituted with the term $\mathbf{B}$. Thus, the rule of α-conversion corresponds to a renaming of the λ-bound variable Y in $\mathbf{A}$. □

One notion that is used frequently within ΩMEGA is that of a term position. Term positions help to identify and single out sub-terms in given terms.

Definition 2.10 (Term position): Let $\mathbb{N}^*$ be the set of words over the set of non-negative integers $\mathbb{N}$ and let ϵ be the *empty word* in $\mathbb{N}^*$. For a term $t \in \mathbf{wff}(\Sigma)$ the *term position* of a distinct sub-term s of t, $p_t(s) \in \mathbb{N}^*$, is inductively defined as follows:

- If $s = t$ then $p_t(s) = \epsilon$,
- if $t = (t_0\, t_1 \ldots t_n)$ and the distinct s occurs in t_i, $0 \leq i \leq n$, then $p_t(s) = i.p_{t_i}(s)$,
- if $t = \lambda x_\blacksquare t'$ and the distinct s is a sub-term of t' then $p_t(s) = 0.p_{t'}(s)$.

We write term positions in brackets as $\langle \pi.\tau \rangle$, where $\pi, \tau \in \mathbb{N}^*$ and '.' denotes the concatenation of words in $\mathbb{N}^*$. □

2.1.2 Semantics

The semantics for ΩMEGA's logic is based on the type system $\mathcal{T}$ that contains as base-type the type of truth values o and the type of individuals ι.

Definition 2.11 (Frame): A *frame* is a collection of nonempty sets $\mathcal{D}_\alpha$, one for each type symbol α such that $\mathcal{D}_o = \{\top, \bot\}$ and $\mathcal{D}_{\alpha\to\beta} \subseteq \mathcal{F}(\mathcal{D}_\alpha, \mathcal{D}_\beta)$, where $\mathcal{F}(\mathcal{D}_\alpha, \mathcal{D}_\beta)$ is the set of all total functions from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$. □

We call the members of $\mathcal{D}_o$ truth values, where $\top$ corresponds to *truth* and $\bot$ corresponds to *falsehood*. The elements of $\mathcal{D}_\iota$ are called individuals.

Definition 2.12 (Interpretation of constants): Given a frame $\mathcal{D}_\alpha$ and a signature Σ with respect to $\mathcal{T}$. We call the typed function $\mathcal{I} : \Sigma \to \mathcal{D}$ an *interpretation of constants* (or simply *interpretation*) with *support* $\mathcal{D}$. □

With the help of the interpretation function $\mathcal{I}$ it is now possible to give meaning to the logical constants we have introduced in definition 2.4.

Definition 2.13 (Interpretation of logical constants): Given the logical constants $\{\neg, \lor, \Pi^\alpha, \imath^\alpha\} \subseteq \Sigma$ from definition 2.4, we restrict the interpretation $\mathcal{I}$ in the following way:

(i) $\mathcal{I}(\neg)(d) = \top$ if an only if $d = \bot$, $d \in \mathcal{D}_o$

(ii) $\mathcal{I}(\lor)(d, e) = \top$ iff $d = \top$ or $e = \top$, $d, e \in \mathcal{D}_o$

(iii) $\mathcal{I}(\Pi^\alpha)(d) = \top$ iff $d(a) = \top$ for all $a \in \mathcal{D}_\alpha$ with $d \in \mathcal{D}_{\alpha\to o}$

(iv) $\mathcal{I}(\imath^\alpha)(d) = c$ iff $d = \{c\}$ for $d \in \mathcal{D}_{\alpha o}$ and $c \in \mathcal{D}_\alpha$ □

In point *(iii)* of the preceding definition the notation $d(a)$ stands for the application of the function $d \in \mathcal{D}_{\alpha\to o}$ to the object $a \in \mathcal{D}_\alpha$ as mentioned in 2.7.

Although the logical constants from definition 2.13 are sufficient to define a proper logic, for notational convenience we enrich our signature by addition of the following abbreviations[1]:

- the *universal quantifier* $\forall_{\alpha o o}$ such that $\forall X_\alpha \mathbf{.} \mathbf{A}_o := \Pi^\alpha(\lambda X_\alpha \mathbf{.} \mathbf{A})$
- the *existential quantifier* $\exists_{\alpha o o}$ such that $\exists X_\alpha \mathbf{.} \mathbf{A}_o := \neg(\forall X \mathbf{.} \neg\mathbf{A})$
- the *conjunction* $\land_{ooo}$ such that $\mathbf{A}_o \land \mathbf{B}_o := \neg(\neg\mathbf{A} \lor \neg\mathbf{B})$
- the *implication* $\Rightarrow_{ooo}$ such that $\mathbf{A}_o \Rightarrow \mathbf{B}_o := \neg\mathbf{A} \lor \mathbf{B}$
- the *equivalence* $\Leftrightarrow_{ooo}$ such that $\mathbf{A}_o \Leftrightarrow \mathbf{B}_o := (\mathbf{A} \Rightarrow \mathbf{B}) \land (\mathbf{B} \Rightarrow \mathcal{A})$
- the *equality* $\doteq_{\alpha\alpha o}$ such that $\mathbf{M}_\alpha \doteq \mathbf{N}_\alpha := \forall P_{\alpha o} \mathbf{.} P(\mathbf{M}) \Rightarrow P(\mathbf{N})$

The given definition of equality corresponds to the definition of LEIBNIZ equality. In order to avoid confusion we shall write equality in formulas as $\doteq$ throughout this chapter, however, in the remaining chapters of this thesis equality is again written with the more conventional = symbol. Observe that similar to the definition of Π^α in definition 2.4 the definition of $\doteq^\alpha$ is polymorphic.

[1] In fact, we could define a logic with an even smaller number of logical constants. For instance, ANDREWS defines a higher order logic in [7] using equality and description, only.

So far we are only able to interpret single constants in our semantical domains. Now we will define extensions that cater also for variables and complex formulas.

Definition 2.14 (Variable assignment): Given a frame $\mathcal{D}_\alpha$ and a set of typed variables $\mathcal{V}$ over $\mathcal{T}$ we call a typed function $\varphi : \mathcal{V} \to \mathcal{D}$ a *variable assignment* (or simply *assignment*) with support $\mathcal{D}$. □

Definition 2.15 (Denotation): Let Σ, $\mathcal{V}$ be a signature and a set of variables over $\mathcal{T}$. Let $\mathbf{wff}(\Sigma)$ be the set of well-formed formulas of Σ and let $\mathcal{I} : \Sigma \to \mathcal{D}$ and $\varphi : \mathcal{V} \to \mathcal{D}$ be the corresponding interpretation and assignment, respectively, then we define the *denotation* $\mathcal{I}_\varphi : \mathbf{wff}(\Sigma) \to \mathcal{D}$ inductively as:

(i) $\mathcal{I}_\varphi(X) = \varphi(X)$, if $X \in \mathcal{V}$

(ii) $\mathcal{I}_\varphi(c) = \mathcal{I}(c)$, if $c \in \Sigma$

(iii) $\mathcal{I}_\varphi(\mathbf{AB}) = \mathcal{I}_\varphi(\mathbf{A})(\mathcal{I}_\varphi(\mathbf{B}))$

(iv) $\mathcal{I}_\varphi(\lambda X_\alpha \blacksquare \mathbf{A}_\beta)$ as the function in $\mathcal{D}_{\alpha\beta}$ such that for all $z \in \mathcal{D}_\alpha$ holds: $(\mathcal{I}_\varphi(\lambda X_\alpha \blacksquare \mathbf{A}))z := \mathcal{I}_{\varphi,[z/X]}(\mathbf{A})$. □

Given our definition of a frame so far, we cannot be sure that the function required in definition 2.15 (iv) exists. The domain $\mathcal{D}_{\alpha\beta}$ might be too sparse [4]. Because of the inductive nature of the definition this problem also affects 2.15 (iii). However, in the semantical domains of interest — the Henkin models [108] — this possibility is explicitely excluded; that is, every formula in $\mathbf{wff}(\Sigma)$ can be denoted.

Definition 2.16 (Henkin models): Let $\mathcal{I}_\varphi : \mathbf{wff}(\Sigma) \to \mathcal{D}$ be a denotation such that $\mathcal{I}_\varphi$ is defined for each formula $\mathbf{A} \in \mathbf{wff}(\Sigma)$, then we call the pair $\mathbf{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ a *Henkin model* for $\mathbf{wff}(\Sigma)$. □

Being certain that every formula in $\mathbf{wff}(\Sigma)$ can actually be denoted it is now possible to exactly evaluate propositions.

Definition 2.17: Let $\mathbf{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ be a Henkin model and $\mathbf{P} \in \mathbf{wff}_o(\Sigma)$ be a proposition then we have:

(i) $\mathbf{P}$ is *valid in the model* $\mathbf{M}$ when for each assignment φ holds that $\mathcal{I}_\varphi(\mathbf{P}) = \top$.

(ii) $\mathbf{P}$ is called *valid* or a *tautology* if $\mathbf{P}$ is true in each Henkin model $\langle \mathcal{D}, \mathcal{I} \rangle$.

(iii) Given a set of propositions Γ we say that Γ is *satisfiable* in $\mathbf{M}$, provided there is some assignment φ such that $\mathcal{I}_\varphi(\mathbf{P}) = \top$ for all $\mathbf{P} \in \Gamma$.

(iv) A proposition $\mathbf{P}$ *follows semantically* from a set of propositions Γ if $\mathbf{P}$ is valid in each Henkin model $\langle \mathcal{D}, \mathcal{I} \rangle$ in which the elements Γ are valid. □

Notation 2.18: To simplify the notation given in definition 2.17 we shall write $\Gamma \models \mathbf{P}$ to indicate that $\mathbf{P}$ follows semantically from the set of propositions Γ and $\models \mathbf{P}$ if $\mathbf{P}$ is a tautology.

The Henkin models given in definition 2.16 are also called *generalized models* since they still allow for incomplete domains (even with the restriction we discussed with respect to definition 2.15):

$$\mathcal{D}_{\alpha\to\beta} \subseteq \mathcal{F}(\mathcal{D}_\alpha \mathcal{D}_\beta). \tag{2.1}$$

This means that all formulas valid in a Henkin model are only a subset of all possibly valid formulas. Based on the notion of Henkin models we can define the *standard models* by requiring

$$\mathcal{D}_{\alpha\to\beta} = \mathcal{F}(\mathcal{D}_\alpha, \mathcal{D}_\beta). \tag{2.2}$$

The standard models are thus a subset of the Henkin models. But the set of valid formulas contained in an arbitrary Henkin model is generally smaller then the set of valid formulas in the standard models. However, GÖDEL could show in his *incompleteness theorem* that there do not exist calculi that are both sound and complete for standard models, whereas it was proved by HENKIN in 1950 that complete and sound calculi can be constructed for Henkin models.

In this thesis we will be concerned neither with the theoretical consequences of this fact nor with completeness considerations of calculi. Instead we refer to [7, 20] for a more detailed introduction and examination of this subject.

2.1.3 Calculus

The original *natural deduction* (ND) calculus was introduced by GENTZEN [95] in 1935. The idea is to model mathematical problem solving behavior in small logical steps for a basic first order logic. Thereby a *theorem* is derived from a given set of *hypotheses* by successively applying *inference rules*. In this section we introduce ΩMEGA's variant of GENTZEN's classical ND calculus, which also caters for the higher order particularities.

For the definition of ΩMEGA's ND calculus we assume the higher order language defined in the previous sections. In particular, we presuppose the semantics of our logical constants to be as given in definition 2.13 and to have the subsequently defined abbreviations available. Although confining ourselves to the original logical constants from definition 2.4 would give a leaner calculus, we prefer a more expressive and intuitive basic calculus by also having inference rules for the abbreviations available. However, the larger the basic calculus is, the less efficient it is to check complete proofs automatically. Therefore, we will not allow for equality and equivalence as primitive concepts and rather define them as derived concepts as given in section 2.2.1.1.

Before defining the single calculus rules we introduce a tree notation to denote the rules of inference.

Definition 2.19 (Proof trees): Let $A_1, \ldots, A_n, A, B \in \mathbf{wff}_o(\Sigma)$ be propositions, we call a *proof tree* one of the following:

(i) $[A]$ where A is a *hypothesis*

(ii) $\frac{}{B}\,\mathcal{R}$ for the *inference rule* $\mathcal{R}$. We call B *conclusion* and $\mathcal{R}$ an *initial rule*

(iii) $\frac{A_1 \ \ldots \ A_n}{B}\,\mathcal{R}$ if B follows from $A_1, \ldots, A_n$ by application of the *inference rule* $\mathcal{R}$. We call $A_1, \ldots, A_n$ *premises*.

(iv) $\begin{matrix}[A]\\ \vdots\\ B\end{matrix}$ if B can be derived from A in a finite number of *inference steps* (i.e., applications of inference rules).

□

We now define the inference rules of ΩMEGA's ND-calculus. Basically we have one introduction and elimination rule for each logical connective and each quantifier. For the elimination of conjunctions and for the introduction of disjunctions we have two symmetrical rules, respectively. Additionally, there is one rule for eliminating of falsehood (ex falso quod libet). While all these rules are basically first order we have also one proper higher order rule that performs λ conversions.

Definition 2.20 (Inference rules): Given propositions $P, Q, R \in \mathbf{wff}_o(\Sigma)$ we can define the inference rules of the natural deduction calculus as follows:

$$\frac{\bot}{P}\ \bot_E$$

$$\frac{P \quad \neg P}{\bot}\ \neg_E \qquad\qquad \frac{\begin{array}{c}[P]\\ \vdots\\ \bot\end{array}}{\neg P}\ \neg_I$$

$$\frac{P \wedge Q}{P}\ \wedge_{El} \quad \frac{P \wedge Q}{Q}\ \wedge_{Er} \qquad\qquad \frac{P \quad Q}{P \wedge Q}\ \wedge_I$$

$$\frac{P \vee Q \quad \begin{array}{c}[P]\\ \vdots\\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots\\ R\end{array}}{R}\ \vee_E \qquad\qquad \frac{P}{P \vee Q}\ \vee_{Ir} \quad \frac{Q}{P \vee Q}\ \vee_{Il}$$

$$\frac{P \quad P \Rightarrow Q}{Q}\ \Rightarrow_E \qquad\qquad \frac{\begin{array}{c}[P]\\ \vdots\\ Q\end{array}}{P \Rightarrow Q}\ \Rightarrow_I$$

$$\frac{\forall x. P}{[t/x]P}\ \forall_E(t) \qquad\qquad \frac{[t/x]P}{\forall x. P}\ \forall_I(t) \text{ with } t \text{ new in } P$$

$$\frac{\exists x. P \quad \begin{array}{c}[t/x]P\\ \vdots\\ Q\end{array}}{Q}\ \exists_E(t) \text{ with } t \text{ new} \qquad\qquad \frac{[t/x]P}{\exists x. P}\ \exists_I(t)$$

$$\frac{A}{B}\ \lambda\!\leftrightarrow$$

In the rules for the quantifiers $[t/x]P$ means that the term t is substituted for all occurrences of the variable x in P. The substituted term t is given in parentheses behind the rule name and is called a *parameter* of the rule. The $\forall_I$ and $\exists_E$ rules have *eigenvariable conditions* that require that the term t does not already occur in the proposition P in case of the $\forall_I$ rule. In the $\exists_E$ rule the term t must not occur anywhere else in the proof. □

The $\lambda\!\leftrightarrow$ rule is the higher order rule that allows to close a goal with a support that is equal with respect of the λ-conversions given in definition 2.9; that is, A denotes the same term as B up to $\beta\eta$-reduction and renaming.

In addition to the inference rules ΩMEGA's ND-calculus also has some *axioms* in order to be complete. We have one axiom to ensure that there exist exactly two truth values (i.e., that we have a classical logic), two axioms for extensionality and one axiom for the description operator.

Definition 2.21 (Axioms): We define the following four axioms for our calculus:

- $\forall A_o\centerdot A \lor \neg A$ (Tertium non datur)
- $\forall M_{\alpha\beta}\centerdot \forall N_{\alpha\beta}\centerdot [\forall X_\alpha\centerdot MX \doteq NX] \Rightarrow [M \doteq N]$ (Functional extensionality)
- $\forall A_o\centerdot \forall B_o\centerdot (A \Leftrightarrow B) \Rightarrow (A \doteq B)$ (Boolean extensionality)
- $\forall P_{\alpha o}\centerdot \exists X_\alpha\centerdot [PX \land \forall Y_\alpha\centerdot PY \Rightarrow [X = Y]] \Rightarrow P\imath_{\alpha o\alpha}P$ (Description)

□

The axiom of description in the preceding definition gives us a more precise understanding of the description operator as a partial function that acts only on singleton sets. It expresses that for every set $P_{\alpha o}$ that contains exactly one unique element, the description operator applied to the set P returns an element of P, which is, of course, its only element. It can be shown that a description operator needs to be defined and axiomatized only for the base type ι and subsequent description operators for higher types can then be derived. However, in ΩMEGA we adopted a uniform view on all description operators by axiomatizing them for all types $\alpha \in \mathcal{T}$. For a introduction to the description operator and its properties see [5].

The two axioms of extensionality could also be formulated as equivalences. However, since equality is defined via Leibniz equality in ΩMEGA the respective reverse directions can be infered within the calculus and were thus omitted. Naturally, the given axioms could have been integrated into the calculus by defining appropriate rules. However, in order to keep the calculus lean we have rather chosen the axiomatic approach in ΩMEGA. Moreover, it did not seem desirable to have basic calculus rules containing concepts such as equality or equivalence, which in turn can be replaced by their respective definitions (see also the discussion in section 2.2.1.1).

Definition 2.22 (Natural deduction proof): Given a set of propositions $\mathcal{H} \subset \mathbf{wff}_o(\Sigma)$ and a proposition $F \in \mathbf{wff}_o(\Sigma)$, a *natural deduction proof* for F under the assumption of $\mathcal{H}$ is a finite sequence of inference rule applications that derives F from $\mathcal{H}$. We write $\mathcal{H} \vdash_{ND} F$ or simply $\mathcal{H} \vdash F$. We call $\mathcal{H}$ the *hypotheses* or *assumptions* of the proof and F the *theorem* or *conclusion*. □

At this point we observe that our calculus defined so far does not contain any means to introduce *cuts* into a derivation. Although it has been shown by TAKAHASHI [200, 201] that *cut-elimination* holds for higher order calculi with extensionality, it is still an open problem whether appropriate cut-elimination algorithms terminate. (See also [168] for a discussion on cut-elimination in type theory.) A possible cut rule for our natural deduction calculus is of the form

$$\frac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C} \quad ,$$

which is essentially *modus barbara*. Indeed ΩMEGA offers a way to introduce cuts by having modus barbara as a tactic available (see section 2.2.3 for an introduction of tactics), which can be modeled by a double application of the $\Rightarrow_E$ rule and one application of $\Rightarrow_I$ on the basic calculus level.

Although the tree notation for the ND calculus inference rules is a convenient technique to display the inference rules it is not very practical to denote large proofs.

Thus, in the remainder of this thesis we will present natural deduction proofs in a linearized style as introduced by ANDREWS in [6].

Definition 2.23 (Linearized ND proofs): A *linearized ND proof* is a finite set of proof lines, where each proof line is of the form $L.\ \Delta \vdash F\ (\mathcal{R})$, where L is a unique *label*, $\Delta \vdash F$ is a *sequent* denoting that the formula F can be derived from the set of hypotheses Δ, and $(\mathcal{R})$ is a *justification* expressing how the line was derived in a proof. □

In case there exist lines in the set of proof lines that have not yet been derived from the hypotheses we indicate them with an *open justification*. We call lines with an open justification *open lines* or *open goals* and a set of proof lines containing still open lines a *partial proof*. We call a line that is not open a *closed line*.

We conclude the introduction of ΩMEGA's logic by giving an example of a simple ND proof both in tree and in linearized representation.

Example 2.24:

The linearized natural deduction proof for the assertion:

$$(\forall X_\iota.(P_{\iota o}(X)\Rightarrow Q_{\iota o}(X))\Rightarrow(\forall X_\iota.P(X)\Rightarrow\forall X_\iota.Q(X)))$$

L3.	L3	$\vdash \forall X_\iota.P_{\iota o}(X)$	(Hyp)
L6.	L3	$\vdash P(A_\iota)$	$(\forall E$ L3)
L1.	L1	$\vdash \forall X_\iota.[P(X)\Rightarrow Q_{\iota o}(X)]$	(Hyp)
L7.	L1	$\vdash [P(X_1)\Rightarrow Q(X_1)]$	$(\forall E$ L1)
L5.	L1, L3	$\vdash Q(X_1)$	$(\Rightarrow E$ L6,L7)
L4.	L1, L3	$\vdash \forall X_\iota.Q(X)$	$(\forall I$ L5)
L2.	L1	$\vdash [\forall X_\iota.P(X)\Rightarrow\forall X_\iota.Q(X)]$	$(\Rightarrow I$ L4)
Thm.		$\vdash [\forall X_\iota.[P(X)\Rightarrow Q(X)]\Rightarrow[\forall X_\iota.P(X)\Rightarrow\forall X_\iota.Q(X)]]$	$(\Rightarrow I$ L2)

The same proof in tree representation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall X_\iota.P(X)]^2}{P(X_1)}\forall E \qquad \dfrac{[\forall X_\iota.(P(X)\Rightarrow Q(X))]^1}{(P(X_1)\Rightarrow Q(X_1))}\forall E}{Q(A_\iota)}\Rightarrow E}{\forall X_\iota.Q(X)}\forall I}{(\forall X_\iota.P(X)\Rightarrow\forall X_\iota.Q(X))}\Rightarrow I^2}{(\forall X_\iota.(P_{\iota o}(X)\Rightarrow Q_{\iota o}(X))\Rightarrow(\forall X_\iota.P(X)\Rightarrow\forall X_\iota.Q(X)))}\Rightarrow I^1$$

Note that the superscript numbers indicate which hypotheses was introduced during which rule application.

2.2 Constructing Proofs in ΩMEGA

Although ΩMEGA's purpose is to help proving theorems in the natural deduction calculus introduced in the preceding section, the proof construction itself is not necessarily carried out in the basic calculus. Instead, proofs are generally constructed on a more abstract level. In particular, a user can employ interactive tactical theorem proving, automatic proof planning or the application of external systems. Furthermore, proofs in ΩMEGA are always constructed within the *context of a mathematical theory*. Different mathematical theories are stored in ΩMEGA's knowledge base and provide — among other things — defined concepts, their axiomatization, and already proved theorems, that can be incorporated into proofs. Thus, proofs in ΩMEGA are actually constructed with respect to given background

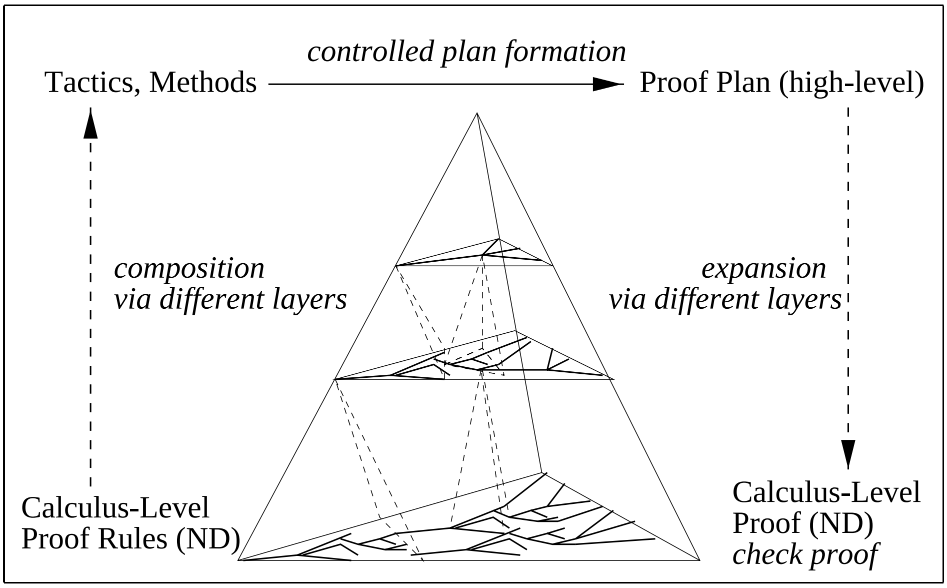


Figure 2.1: Proof Plan Data Structure.

theory in a generalized natural deduction calculus where justifications can be ND rules and also abstract tactics or proof methods as well as applications of external systems.

However, for a proof to be valid in ΩMEGA it needs to be refined into a calculus level natural deduction proof. Therefore, abstract justifications have to be expandable into calculus level subproofs. This expansion can be hierarchical, meaning that the expanded subproof may again contain abstract justifications, that have to be expanded. All abstract levels of a proof as well as its calculus level are stored in a single *proof plan data structure* ($\mathcal{PDS}$) [60], which constitutes ΩMEGA's actual proof object. We call abstract inference steps *planned* since they may contain inference steps that can sometimes be faulty. Hence, the expansion of such a step can fail, leaving a part of the proof still open. This feature permits us to employ uncertain heuristics and external reasoners that are not necessarily always correct. But of course we pay for this extra freedom with the price to proof check every final proof.

Figure 2.1 depicts schematically the composition of the $\mathcal{PDS}$, which is represented as an acyclic graph comprising proof nodes at different levels of abstraction. The abstractions are realized via the justifications of the single nodes; that is, a node can have an *abstract justification* at an upper layer in the $\mathcal{PDS}$ which corresponds to a partial proof at a lower layer. Note that the formulas of the particular nodes involved stay the same on all levels of abstraction. Thus, the $\mathcal{PDS}$ allows for derivational abstraction but not for abstraction of the objects of our logic.

Once a proof is constructed all abstract justifications have to be expanded in order to gain an ND calculus level proof. This proof is then machine-checkable with ΩMEGA's proof checker and its correctness relies solely on the correctness of the verifier and the underlying calculus. A concrete example of an abstract proof step and its expansion is given in section 2.2.3.

For a given *theorem* and its *hypotheses* a proof is constructed by successively applying *inference rules*, which are either abstract proving steps or ND calculus rules and the reasoning may be either backwards or forwards. In the former case, rules are applied to the theorem, resulting in the introduction of the premises of the rule as new open goals. If an applied rule has more then one premises the problem is split into several subproblems, which have to be shown. In the latter, rules are applied to the hypotheses and the conclusions of the rule are introduced as new nodes into the proof. These nodes become new *supports* of those goals that depended on the hypotheses the rule was applied to. For many applications it is interesting to mix forward and backward reasoning and this kind of *middle-out reasoning* is therefore supported as well.

In the sequel we introduce the different types of inference rules that can be applied to construct proofs in ΩMEGA. We also give examples for the different possible application directions and explain how possible expansions are performed and what they look like.

2.2.1 Knowledge Base

Proofs in ΩMEGA are constructed with respect to a knowledge base of mathematical facts. These facts are organized into a hierarchy of theories, which are connected by a simple inheritance mechanism. Single theories contain defined concepts that enable a compact problem formulation and axioms and theorems that can be directly incorporated and applied in a proof. Furthermore the knowledge base permits the introduction of a sort concept for proofs. This, however, is not a full grown sort concept as given in the literature (for instance by SCHMIDT-SCHAUSS in [183] in the context of first order logic and by KOHLHASE for higher order logic in [129]), instead it is a conservative extension to the logic given in the preceding section by simply allowing for sorted quantifications.

2.2.1.1 Definitions

Definitions in ΩMEGA have the same role as definitions in a mathematical textbook: They help to shorten formulas and proofs by introducing abbreviations for complex concepts. A definition is generally given as a λ-term and can be expanded if necessary.

We have already seen two defined concepts, namely *equality* and *equivalence* in this chapter. Their respective definitions in the knowledge base are of the form

$$\begin{array}{rcl} \doteq_{\alpha\alpha o} & \equiv & \lambda x_{\alpha \bullet} \lambda y_{\alpha \bullet} \forall P_{\alpha o \bullet} P(x) \Rightarrow P(y) \quad \text{and} \\ \Leftrightarrow_{ooo} & \equiv & \lambda a_{o \bullet} \lambda b_{o \bullet} (a \Rightarrow b) \wedge (b \Rightarrow a). \end{array}$$

Here $\equiv$ is the *definition symbol*, meaning that the symbol on the lefthand side (the defined symbol) is an abbreviation for the λ-term on the righthand side. Other concepts in ΩMEGA's knowledge base are, for instance, a basic notion of set theory, such as the element property or the union of two sets, which are defined as

$$\begin{array}{rcl} \in_{\alpha\alpha o} & \equiv & \lambda x_{\alpha \bullet} \lambda P_{\alpha o \bullet} P(x) \quad \text{and} \\ \cup_{(\alpha o)(\alpha o)o} & \equiv & \lambda U_{\alpha o \bullet} \lambda V_{\alpha o \bullet} \lambda x_{\alpha \bullet} U(x) \vee V(x). \end{array}$$

To illustrate the concept of definition expansion consider the term $y_\iota \in A_{\iota o} \cup B_{\iota o}$, which states that y is a member of the union of the sets A and B. Writing this in prefix notation we get $(\in y(\cup AB))$. Replacing both the element and the union symbol by their respective definitions we get $((\lambda x_{\iota \bullet} \lambda P_{\iota o \bullet} P(x))y((\lambda U_{\iota o \bullet} \lambda V_{\iota o \bullet} \lambda x_{\iota \bullet} U(x) \vee V(x))AB))$. Applying β-reduction to this term yields $A(y) \vee B(y)$, which corresponds to the simple proposition that y is either in A or in B.

L_1.	L_1	$\vdash \forall A_o \blacksquare A \vee \neg A$	(Axiom)
L_2.	L_1	$\vdash P \vee \neg P$	$(\forall_E\ L_1\ P)$
L_3.	L_3	$\vdash P \Rightarrow Q$	(Hyp)
L_4.	L_4	$\vdash P$	(Hyp)
L_5.	L_3, L_4	$\vdash Q$	$(\Rightarrow_E\ L_4\ L_3)$
L_6.	L_3, L_4	$\vdash \neg P \vee Q$	$(\vee_{Ir}\ L_5)$
L_7.	L_7	$\vdash \neg P$	(Hyp)
L_8.	L_7	$\vdash \neg P \vee Q$	$(\vee_{Il}\ L_4)$
L_9.	L_3	$\vdash \neg P \vee Q$	$(\vee_E\ L_2\ L_6\ L_8)$
L_{10}.		$\vdash (P \Rightarrow Q) \Rightarrow (\neg P \Rightarrow Q)$	$(\Rightarrow_I\ L_9)$

Table 2.1: Proof involving the axiom of the excluded middle.

2.2.1.2 Axioms and Theorems

Axioms in ΩMEGA's knowledge base are facts stated without proof. Examples are the three axioms given in definition 2.21 that cannot be derived in ΩMEGA's calculus. Apart from these, certain defined concepts need to be axiomatized.

Theorems on the other hand are facts in the knowledge base for which a valid proof has already been derived in ΩMEGA. A trivial theorem contained in the knowledge base is, for instance, $\forall a_o \blacksquare (\bot \wedge a) \Leftrightarrow \bot$. Every problem in ΩMEGA for which a valid proof (i.e., a fully expanded ND proof that has been successfully machine-checked) has been derived can be stored as a theorem in the knowledge base.

During proof construction in ΩMEGA both theorems and axioms can be directly imported into the proof as so-called *theory assertions* or simply *assertions*. Assertions are applied like any hypotheses of the proof, however, in case the imported assertion depends itself on additional assumptions, these assumptions have to be shown to hold. In other words if the imported assertion is of the form $\mathcal{P}_1 \wedge \ldots \wedge \mathcal{P}_n \Rightarrow \mathcal{T}$ the assumptions $\mathcal{P}_1, \ldots, \mathcal{P}_n$ become new subgoals that need to be shown.

An example of a proof involving the application of the tertium non datur axiom is given in table 2.1. The proposition to prove is $(P \Rightarrow Q) \Rightarrow (\neg P \Rightarrow Q)$ given in line L_{10}. The axiom is imported into the proof in line L_1 — as indicated by the justification *Axiom* — and is treated similarly to the other hypotheses that originate from the application of the $\Rightarrow_I$ and $\vee_E$ rules.

2.2.1.3 Light Sorts

Sorted logics incorporate knowledge about the terms into the logic; that is, terms are annotated with semantical information and certain operations, such as unification, term substitution, etc. can be performed only between terms of the same sort. Moreover, sort systems can be enhanced by having hierarchies of sub-sorts as well. Sorted logics with a flat sort structure are called *many sorted logics*, whereas a logic that supports a hierarchy on the sort structure is called an order sorted logic. This is a powerful mechanism, which enhances the expressiveness of a logic and has often drastic consequences (e.g., on the unification type of that logic).

In the context of a typed higher order logic sorts are a refinement of the type system. For instance, terms denoting non-negative integers can be labeled with a sort natural, which in turn can be a sub-sort of the integers. Functions between integers can then be tagged with an appropriate sort, as well.

The use of sorts can enhance the readability of formulas of a logic. Moreover, for automation purposes, sorts can drastically reduce the search space. Effective sorts for first order and higher order logics are discussed in [183] and [129], respectively.

ΩMEGA's sort concept is, however, less elaborate. Instead of having a full fledged sort system, ΩMEGA only permits the use of so-called *light sorts*; that is, quantified variables are defined with respect to a set, which gives the range of the possible instantiations of the variable. This set is treated as the sort of the variable. Once the variable is instantiated the sort information is explicitly introduced into the proof and, if necessary, has to be explicitly justified.

Thus, the actual sorts are introduced as attachments of the two quantifiers $\forall$ and $\exists$, which we shall write in this thesis as $\forall x_\alpha{:}_{M_{\alpha o}}$ and $\exists y_\alpha{:}_{M_{\alpha o}}$, indicating that x and y are in the set M. Each sorted quantifier is, of course, only an abbreviation for a more complex expression as we can observe with the following two expressions:

(i) $\forall x_\alpha{:}_{M_{\alpha o}}\text{.} P_{\alpha o}(x)$ abbreviates $\forall x_\alpha\text{.}[x \in M_{\alpha o}]\Rightarrow P_{\alpha o}(x)$

(ii) $\exists y_\alpha{:}_{M_{\alpha o}}\text{.} Q_{\alpha o}(y)$ abbreviates $\exists y_\alpha\text{.}[y \in M_{\alpha o}] \wedge Q_{\alpha o}(y)$

Using light sorts in ΩMEGA has two advantages: On the one hand the term construction is kept decidable; note that this is no longer guaranteed in a logic with both polymorphic types and subsorts. On the other hand, light sorts add to the readability of the logic since they allow to state formulas of theorems and problems more concisely. As an example of the latter consider the following statement for integers

$$\forall x{:}_{\mathbb{Z}}\text{.} \exists y{:}_{\mathbb{Z}}\text{.} (x + y) \doteq 0,$$

which is relatively concise using sorted quantifiers. It becomes much less readable if we abolish abbreviations:

$$\forall x\text{.} [x \in \mathbb{Z}] \Rightarrow [\exists y\text{.} [y \in \mathbb{Z}] \wedge [(x + y) \doteq 0]].$$

During proof construction sorted quantifiers have to be treated slightly differently than their unsorted counterparts. This treatment is given in more detail in section 2.2.3.

2.2.2 Calculus Rules

The rules in ΩMEGA are essentially the natural deduction rules given in definition 2.20. Additionally we have an initial rule to introduce facts from the knowledge base as for instance in the proof in table 2.1. The respective justification then depends on the type of imported assertion, whether it is an axiom, a theorem, or a lemma. Furthermore, there exist the following three rules:

$$\frac{A}{A}\ Weaken \qquad \frac{A}{[t'/t]B} \equiv_E(t \equiv t', \pi) \qquad \frac{[t'/t]A}{B} \equiv_I(t \equiv t', \pi)$$

Weaken is a special case of the $\lambda\leftrightarrow$ rule since it allows to justify a goal with a support node containing the same formula meaning they are trivially equal with respect to λ-conversion. The latter two rules, $\equiv_E$ and $\equiv_I$, deal with the elimination and introduction of definitions from the knowledge base. The notation $[t'/t]B$ means that the occurrence of the defined concept t at sub-term position π in B is replaced by its definition t'. Both the actual definition and the term position are given as parameters of the rules. However, we usually give only the definiens (i.e., the lefthand side of a definition) as a parameter in the justification.

Most rules can be applied in a *forward* and *backward* direction. Certain rules can also be applied *sideways* and for *closing* subproofs. For example the *modus ponens* rule $\Rightarrow_E$ given as $\dfrac{P \quad P{\Rightarrow}Q}{Q}$ can be applied in five different directions: (i) Forwards, where P and $P{\Rightarrow}Q$ are given and Q is introduced as a new closed line. Three sideways directions (ii) only $P{\Rightarrow}Q$ is given, then Q is introduced as a new closed line and P as a new open line, (iii) $P{\Rightarrow}Q$ and Q are given and P is introduced and (iv) P and Q are given and the implication is introduced as new open goal. Finally, closing the subproof, if (v) all three terms are given, then the open goal Q is closed. $\Rightarrow_E$ cannot be applied in a backward direction. However, an instance of a rule that can be applied backwards only is $\Rightarrow_I$ as given in definition 2.20. Applied to an open goal containing an implication the succedent of the implication is introduced as a new open goal, whereas the antecedent is introduced as a new hypothesis. However, this hypotheses is a *local hypothesis* since it becomes only an additional support of the newly introduced goal and will not become a hypotheses for all other, already existing goals.

Since rules form the lowest level of the $\mathcal{PDS}$ they have no expansion; that is, once a line is justified by an ND rule, it cannot be mapped to a more fine grained level.

2.2.3 Tactics

Many interactive systems use tactical theorem proving for complex and more human oriented proofs (cf. NUPRL [65], ISABELLE [166]). Tactical theorem proving is based on the notion of a *tactic*, which encapsulates repeatedly occurring sequences of inference steps into macro-steps. Tactics are built in a bottom-up fashion by combining sequences of calculus rule applications using so-called *tacticals* (see [101]). Tacticals can also be used to combine already defined tactics to even more complex tactics. The application of a tactic results in a sequence of calculus rules. And since every tactic is only a combination of calculus rules and immediately expands to that level after application, it is *a priori* correct, given the correctness of the underlying calculus.

In ΩMEGA, however, we favor a top-down approach for constructing tactics and thereby deliberately give up the guaranteed correctness.[2] A tactic in ΩMEGA is a procedure that performs a derivation and whose application in a proof corresponds to a single inference step. It can thus be seen as a generalized form of a calculus rule and we state tactics in the same proof tree form containing premises, conclusions, and possibly hypotheses and parameters. But unlike ND-calculus rules tactics can have multiple conclusions.

As the application of a tactic in ΩMEGA is not immediately decomposed into a sequence of single calculus rule steps, correctness has to be ensured *a posteriori*. This is done by expanding a proof step whose justification contains a tactic application. This expansion process subsequently introduces a more fine-grained proof plan justifying the derivation of the tactic at a conceptually more detailed layer. The expansion can be recursive in the sense that the introduced proof plan can again contain abstract inference steps, which have to be expanded in turn. If the expansion is successful, the original tactic application has been transformed into a calculus level proof, which can be machine-checked. This way of dealing with tactics permits us to employ uncertain heuristics within tactics whose expansions might fail occasionally.

[2] Therefore, ΩMEGA's tactics are also sometimes called *failing tactics*. We shall, however, use the term *tactic* throughout the thesis for simplicity.

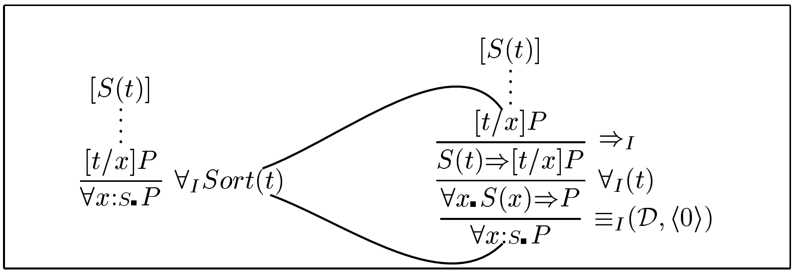


Figure 2.2: Expansion of the $\forall_I Sort$ tactic.

As an example of a tactic and its expansion consider the $\forall_I Sort$ tactic as given on the lefthand side in figure 2.2. $\forall_I Sort$ is one of four tactics for the treatment of the sorted quantifiers introduced in section 2.2.1.3. Its purpose is similar to that of the $\forall_I$ rule and likewise it has an *eigenvariable* condition on its parameter, the term t, it introduces. But additionally $\forall_I Sort$ treats the sort of the quantified variable by adding an appropriate hypotheses, stating that the newly introduced term t is in the set S.

When $\forall_I Sort$ is expanded, the calculus level proof given on the righthand side of figure 2.2 is introduced. The proof now consists of three steps instead of one: First, the definition of the sorted universal quantifier is rewritten with a $\equiv_I$ rule. Here the first parameter standing for the actual definition is only a substitute to preserve space. Hence $\mathcal{D}$ corresponds to the definition of the sorted universal quantifier given in section 2.2.1.3. The second parameter is the position at which the defined concept occurs, in our case this is the position $\langle 0 \rangle$. In the second step the actual $\forall_I$ application takes place leaving us with an implication that is subsequently split with an $\Rightarrow_I$ rule.

In addition to the $\forall_I Sort$ tactic there exist three more tactics to deal with sorted quantifiers, namely $\forall_E Sort$, $\exists_I Sort$, and $\exists_E Sort$. Their expansions work all similar to the one given here; that is, after introducing the definition of the respective sorted quantifier, the resulting formula is appropriately split.

2.2.4 Proof Planning

Proof planning was originally conceived as an extension of tactical theorem proving to automate theorem proving at the more abstract level of tactics. The key idea of BUNDY [52] is to augment individual tactics with pre- and postconditions. This results in planning operators or so called *methods*. A mathematical theorem is then considered as a *planning problem*. A planning problem in artificial intelligence consists of an *initial world state* describing some initial situation, a *goal world state* that describes a desired situation, and a set of operators which describe actions that can change the world state. The planning problem is then to compute a sequence of operator applications that transform the initial state into a goal state. In proof planning the initial state consists of the proof assumptions and the goal state consists of the theorem. We apply artificial intelligence planning techniques to search for a sequence of methods that derives the theorem from the assumptions.

2.2.4.1 Knowledge Based Proof Planning

In the ΩMEGA system the traditional proof planning approach is enriched by incorporating mathematical knowledge into the planning process (see [156] for de-

Method: $\exists_I Resclass$				
Premises	$\oplus L_3, \oplus L_1$			
Appl. Cond.	$ResclassSet(RS_n, n, N_{set})$			
Conclusions	$\ominus L_5$			
Declarative Content	(L_1)	Δ	$\vdash mv \in N_{set}$	$(Open)$
	(L_2)	Δ	$\vdash c \in RS_n$	$(ConResclSet\ L_1)$
	(L_3)	Δ	$\vdash P[cl_n(mv)]$	$(Open)$
	(L_4)	Δ	$\vdash P[c]$	$(ConRescl\ L_3)$
	(L_5)	Δ	$\vdash \exists x{:}_{RS_n}. P[x]$	$(\exists_I Sort\ L_2\ L_4)$

Figure 2.3: An example of a method.

tails). This is motivated by the fact that mathematicians rely on domain-specific knowledge and are typically experts in a highly specialized field rather than universal experts. In ΩMEGA there are three possibilities to incorporate domain-specific knowledge: Within *methods*, within *control rules*, and within *domain-specific external reasoners* such as computer algebra systems, constraint solvers, or automated theorem provers. Methods in ΩMEGA cannot only encode general proving steps but also steps particular to a mathematical domain. Control rules enable meta-level reasoning about the current proof planning state as well as about the entire history of the proof planning process to guide the search.

We demonstrate the notion of a method and a control rule later on with examples. But first let us sketch briefly ΩMEGA's main planning algorithm, which follows the *precondition achievement planning* paradigm (see, e.g., [78]); that is, the planner tries continuously to reduce open goals by applying a method that has an appropriate postcondition. The method application might then again result in one or more new open goals. Initially, the only open goal is the theorem. During this planning process there are several choice points such as which goal should be tackled or which method should be applied in the next step. These choice points can be influenced by control rules. The planning process ends successfully if there are no more open goals.

Methods A method in ΩMEGA is a data structure that consists of four slots: *Premises*, *conclusions*, *application condition*, and *declarative content*. The declarative content contains a declarative specification of the tactic, which is employed by the method given as a sequence of proof steps. The premise and conclusion slots contain two kinds of information: First, they provide logical information in the sense that the conclusions are supposed to follow logically from the premises by the application of the tactic given in the declarative content. Second, they specify the pre- and postconditions of the method, which are necessary to use the method in planning. In ΩMEGA we denote these pre- and postconditions in a STRIPS-like-notation [84] as add and delete list. When a method is applied then a $\ominus$-conclusion is deleted as an open goal and a $\oplus$-premise is added as a new open goal. Conversely, a $\ominus$-premise is deleted as an assumption and a $\oplus$-conclusion is added as an assumption. Furthermore, the application condition of a method contains additional information on when the method can be legally applied.

Certain methods can be designated as *normalization* or *restriction* methods. The former are methods that perform generic simplification tasks whereas the latter trivially close open subgoals. An example of a normalization method is for instance the $\wedge_E$ method that splits conjunctions in support lines. An instance of a restriction method is the *Weaken* method. The planner can try to apply both normalization

```
(control-rule TryAndErrorStdSelect
              (kind methods)
              (IF (disjunction-supports S))
              (THEN (select ($\forall_I$Resclass ConCongCl
                             $\vee_E^{**}$ $\exists_I$Resclass        ))))
```

Figure 2.4: An example of a control rule.

and restriction methods automatically after each regular method application.

An example for a method is $\exists I$*Resclass* given in figure 2.3, which is a method domain-specific for residue classes. We will see examples of its use in chapter 7. Its purpose is to instantiate an existentially quantified variable over a residue class set with a witness term for which a certain property P holds and to reduce the initial statement on residue classes to a statement on integers. The witness term has to be a concrete element of the residue class set. However, if the method is applied at an early stage of the proof, the planner generally has no knowledge of the true nature of the witness term. Therefore, the method invokes a *middle-out reasoning* [131] process to postpone the actual instantiation; that is, a *meta-variable* is used as temporary substitute for the actual witness term, which will be determined at a later point in the planning process and subsequently instantiated.

$\exists I$*Resclass* is given in terms of the original goal (the conclusion L_5), the two new open goals it produces (the premises L_1 and L_3), and the inference steps deriving L_5 from L_1 and L_3 (given in the declarative content). The method is applicable during the planning process if a current planning goal can be matched against the formula of L_5 and if additionally the application conditions (Appl. Cond.) are satisfied. The condition $ResclassSet(RS_n, n, N_{set})$ is fulfilled if RS_n, the sort of the quantified variable x, qualifies as a residue class set of the form given in chapter 7.1. Its successful evaluation binds the method variables n and N_{set} to the modulo factor of RS_n and the set of integers corresponding to the congruence classes of RS_n, respectively. For instance, the evaluation of $ResclassSet(\mathbb{Z}_2, n, N_{set})$ yields $n \leftarrow 2$ and $N_{set} \leftarrow \{0, 1\}$. The necessary inference steps are indicated by the justifications *ConResclSet* and *ConRescl* in lines L_2 and L_4, which denote tactics that convert statements containing residue class expressions into statements containing the corresponding integer expressions. mv in L_1 and L_3 is a meta-variable that substitutes for the actual witness term.

Control Rules Control rules can be used to influence the proof planner at choice points such as which goal to tackle next or which method to apply to a goal. This is done by restricting given alternative lists (e.g., a list of methods) or by preferring certain elements of an alternative list. This way, alternatives are dynamically restricted or reordered which in turn helps to prune the search space or to promote certain promising search paths. Control rules consist technically of an if- and a then-part. In the if-part predicates about the current proof planning status or the entire planning history are evaluated. In the then-part actions on alternative lists can be executed.

Figure 2.4 gives an example of the control rule `TryAndErrorStdSelect`, which is evaluated at the method selection choice point. It states that if the current goal is supported by a disjunctive support line S the application of the methods $\forall_I$*Resclass*, *ConCongCl*, $\vee_E^{**}$, and $\exists_I$*Resclass* is attempted in this order. The 'select' in the then-part states that all other methods except those specified in the

Strategy: `TryAndError`			
Condition	ResidueClassProperty		
Action	Algorithm	*PPlanner*	
	Parameters	Methods	$\forall_I Resclass$, $ConCongCl$, $\vee_E^{**}$, $\exists_I Resclass$, ...
		C-Rules	`TryAndErrorStdSelect`, ...
		Termination	No-Subgoal

Figure 2.5: An example of a strategy.

control rule are eliminated from the list of alternative methods. Other actions are 'reject' and 'prefer'. The former removes all alternatives specified in the control rule from a given alternative list, whereas the latter reorders the alternative list.

2.2.4.2 Multi-Strategy Proof Planning

Multi-strategy proof planning [155] is an extension of knowledge based proof planning. ΩMEGA's multi-strategy proof planner MULTI enables the specification and combination of a number of planning strategies and to switch flexibly between them during the proof planning process. A strategy can be roughly described as the parametrization of a planning algorithm. Therefore, the basic concepts in MULTI are *algorithms* and *strategies*.

As *algorithm* in MULTI we accept every algorithm that refines or modifies partial proof plans. In particular, the traditional planning facilities are decoupled into three different algorithms: *PPlanner* that introduces method applications, *BackTrack* that backtracks steps, and *InstMeta* that instantiates meta-variables. Each of these algorithms has a set of parameters to influence its behavior. For instance, *PPlanner* has the parameters *methods* and *control rules* to specify the methods and control rules it can employ.

Strategies allow to produce different behaviors of the algorithms by different specifications of their parameters. Technically, a strategy is a condition-action pair. The condition part states for which tasks the strategy is applicable and the action part states which algorithm is employed by the strategy and it also gives the instantiation of its parameters. An example of a parameterization of *PPlanner* is given in figure 2.5. According to its application conditions the `TryAndError` strategy can be applied to goals stating a property of a residue class. For more details on this see chapter 7.

Other parameterizations of *PPlanner* can employ different sets of methods and control rules and can thereby, for instance, lead to proofs conducted with a different proof technique. Different *BackTrack* strategies allow MULTI to perform different types of backtracking, while different parameterization of *InstMeta* can force different behavior during the middle-out reasoning processes.

MULTI enables a flexible combination of different strategies and allows to switch flexibly between strategies during a proof planning process. The selection of different strategies can be influenced by *strategic control rules* similar to the control rules for methods or goals. In particular, MULTI allows for interleaving of strategies; that is, one strategy can produce some subgoals that it cannot solve itself. Then this strategy can be interrupted and other strategies can be invoked on the subgoals. Afterwards, when the subgoals are closed the interrupted strategy can be reinvoked and it continues with its computations.

One major advantage of the multi-strategy proof planning approach is its robustness. Since there can be different strategies to solve the same problem, MULTI might not necessarily fail to prove a problem even if one proof attempt fails. It is also more flexible because reasoning about switching and combining several strategies is possible. Moreover, strategies provide a means for structuring the large amount of method and control rule knowledge as well as to force the planner to apply only certain proof techniques.

2.2.5 External Reasoners

Another method to construct proofs at least partially automatic in ΩMEGA is to delegate certain subproblems to *external reasoners*. An external reasoner does not belong to ΩMEGA's core system, but its functionality can be useful for the task at hand. Examples of external reasoners are for instance automated theorem provers (ATP) and computer algebra systems (CAS).

Some of the external reasoners available in ΩMEGA are

- the first order ATPs OTTER [141], SPASS [209], and PROTEIN [19],
- the higher order ATPs TPS [9] and LEO [26],
- WALDMEISTER [110] and EQP [143] two ATPs based on term rewriting,
- the model generators MACE [140] and SATCHMO [47],
- the constraint solver $\mathcal{CoSIE}$ [222], and
- the computer algebra systems MAPLE [177], GAP [94], and MAGMA [59].

Besides these systems, which can be directly used during the proof construction, there are external systems that are only used for post-processing proofs or for the translation inbetween various proof formats. Most notably of these are the TRAMP system [148] for translating machine-found proofs into the natural deduction proof format and P.REX [83], a system for translating logic calculus proofs into textbook style natural language proofs. Section 4.2 gives a detailed account of the sound integration of computer algebra into ΩMEGA and especially into proof planning.

Automated theorem provers are generally applied to complete subproofs; that is, a subgoal together with its supports is passed on to an ATP. If the prover is successful, the subgoal can be closed, otherwise it remains open. The proof for this subgoal is then given as a proof of the particular external reasoner, which means it is generally in a calculus different to ΩMEGA's calculus. Their integration is achieved using the TRAMP system, which translates the machine-found refutation proofs for instance a proof in the resolution calculus, into ΩMEGA proof plans [117]. These proof plans can, in turn, be expanded to the calculus level. Thus, single proof steps denoting the proof by an automated theorem prover as justification are expanded by translating them into subproofs in natural deduction format.

There is an exception for the two higher order theorem provers TPS and LEO, which can also return partial proofs (i.e., proofs that still contain open subgoals). These can also be incorporated into ΩMEGA and further processed. In particular the TPS integration is relatively straightforward since TPS represents proofs in its own natural deduction style calculus. These rules are simulated as tactics in a special theory in ΩMEGA together with expansion into ΩMEGA's basic calculus. This enables a simple mapping of TPS proofs onto ΩMEGA proof plans [21].

2.3 Summary of Chapter 2

In this chapter we have introduced the basic concepts of the ΩMEGA system. We have defined the natural deduction calculus together with its underlying higher order logic in which proofs are constructed. Moreover, we have seen the main concepts of ΩMEGA's knowledge base that can be used in order to construct proofs.

An important feature of ΩMEGA is the possibility to construct proofs at various levels of abstraction using different interactive and automatic tools. Abstract proofs are considered planned and have to be refined to lower level proofs. All different levels of abstraction of a proof are stored in a single datastructure, the $\mathcal{PDS}$, that also maintains the dependencies between abstract steps and their respective refinements. A proof in ΩMEGA is only valid if it can be fully expanded into a basic calculus level proof that can then be machine-checked using a simple verifier.

Part II

Architecture

Chapter 3

Ω-Ants

In this chapter we introduce the hierarchical blackboard architecture Ω-Ants that is our main means to combine reasoning techniques both for automated and interactive theorem proving. The main idea of the architecture is to distribute search for applicable proof steps into small separate processes that gather as much information as possible about the current proof state. The separation leads to a robust system that, however, can easily absorb an amount of system resources that makes its use more cumbersome than useful. Therefore, we also introduce a resource concept that helps controlling the Ω-Ants's behavior. This makes it possible also to include uncertain and undecidable reasoning techniques and a certain extend of automation. The architecture has been first reported in [27], its resource and knowledge adaptive components have been described in [29, 28] and its automation and parts of its formalization in [31].

The structure of the chapter is as follows: We first motivate the architecture, explain some important preliminary concepts, and introduce its single components. Then we present possible adaptations of the mechanism that depend on both additional knowledge and resource considerations. We shall also introduce possible ways to automate proof search in the mechanism. Furthermore, we give a partial formalization and conduct some theoretical considerations in the penultimate section. Before summarizing the chapter we shall discuss our Ω-Ants architecture with respect to the notions of parallelism in deduction systems, blackboard architectures, and agents as given in the literature.

3.1 Motivation

The original motivation for the architecture presented here was to support users in interactive theorem proving. Interactive theorem provers have been developed in the past to overcome the shortcomings of purely automatic systems by enabling the user to guide the proof search and by directly importing expert knowledge into the system. For large proofs, however, this task might become difficult when the system does not support the user's orientation within the proof and does not provide a sophisticated suggestion mechanism in order to minimize the necessary interactions.

Some interactive verification and theorem proving systems such as VSE [12], TPS [9], PVS [165], or HOL [102] already provide mechanisms for suggesting commands or arguments for commands that apply inference rules. But these mechanisms are usually rather limited in their functionality. Command suggestions for the user are either done by compiling a list of possibly applicable commands directly after a command has been executed or there is an explicit command allowing the user to query the system for a hint. The former approach has been realized, for instance, in the VSE [12] system, where a list of commands is compiled that excludes from the set of available commands all those that are definitely not applicable. Thus, the commands in the resulting list are not necessarily applicable. An example for the latter approach is the tutorial feature of TPS [9] where, on user request, a limited number of commands is tested for applicability and one of them is suggested to the user. Both systems employ very simple test properties in order to be both quick and efficient. This is necessary as these systems are usually mono-threaded; that is, the system can either carry out a computation (such as executing a command or computing suggestions) or accept input from the user but not both at the same time. However while waiting for user input, the system is basically idle.

Once the user chooses a command to execute, a system usually provides certain default values for the single arguments of the command. Here most systems follow a sequential suggestion strategy; that is, they suggest the first argument and after the user input for the first argument, they compute an appropriate suggestion for the second argument and so on. The dependencies for computations to suggest command arguments depend on the implementation, which predetermines the order of the command's argument. This sequential approach is a relict from the times when interactive theorem proving systems were used with text-based interfaces. But when working with graphical user interfaces it is common that all arguments for a single command can be entered in the same widget. Thus, the user can now freely chose the order in which to enter the arguments, and relative to this order default suggestions should be computed for the remaining arguments. Furthermore, there should also be default suggestions provided for all arguments of the command, as soon as the command is selected. This has the effect, that on the one hand suggestions have to be computed for all arguments a priori without any initial user interaction. On the other hand the computations for argument suggestions have to become more flexible since the dependencies may vary. This, however, leads to an explosion of predicates needed to compute argument defaults. While in a sequential way of suggesting arguments we only need n predicates for n arguments of the command, in such a flexible setting we need in the worst case

$$O\left(n \cdot \sum_{i=0}^{n-1} \binom{n-1}{i}\right) = O\left(n \cdot 2^{(n-1)}\right)$$

different predicates to cover all potential dependencies between arguments.

Generally there is a single suggestion computed for each argument of a command, although the command might be actually applicable to several sets of different combination of arguments, from which the user could choose. Moreover, usually only simple computations are performed to determine the argument suggestions, because of two reasons: Firstly, the user should not be kept waiting for too long before a suggestion is made and therefore a quick response is required. And secondly, in a mono-threaded system no further interactions can be made while the suggestions are computed.

In summary, we can say that traditional suggestion mechanisms are limited in their functionality to suggest commands or command parameters since they

(i) assemble a list of possibly applicable commands by excluding those that are definitely not applicable or provide hints only for a very small number of commands.

(ii) use a sequential strategy allowing only for argument suggestions in a particular order.

(iii) only work in interaction with the user and therefore have to be restricted to inexpensive computations as a quick response to the user.

(iv) give exactly one instantiation for each argument of a command, only.

(v) waste computational resources as they do not generate suggestions while there is no user interaction.

The Ω-ANTS mechanism we shall present in this chapter tries to overcome these shortcomings by providing the following features:

(i) Suggestions both for applicable commands and their possible arguments are constantly computed in the background of the system and presented to the user. This allows to incorporate also computationally expensive criteria to check applicability.

(ii) The quality of the suggestions increases with time: Cheap and efficient computations terminate first and more expressive suggestions follow subsequently.

(iii) At any given time the user can execute one of the suggested commands leading to new computations of suggestions for the modified proof context.

(iv) The user can chose from a set of possible arguments suggested for each command.

(v) The user can enter arguments for a command in arbitrary order and appropriate suggestions for the remaining arguments are computed.

This is realized by separating the default suggestion mechanism in most parts from the interactive theorem proving process. For that we use a distributed system based on a two-layered blackboard architecture. On the lower layer we have a society of blackboards, one for each of our commands, whose knowledge sources seek for possible instantiations of arguments of the command in the given proof state. The blackboard itself is a means to both exchange results of the knowledge sources as well as to accumulate sets of possible argument instantiations. Any command for which argument instantiations can be found is possibly applicable in the current proof state and is propagated to the upper layer of the architecture on a single blackboard on which all applicable commands are accumulated and can be presented to the user. On each of the blackboards involved we have sorting criteria that heuristically prefer certain argument instantiations and commands.

The whole distributed mechanism is implemented by providing separate threads for each knowledge source of a blackboard. It runs always in the background of the interactive theorem proving environment thereby constantly producing command suggestions that are dynamically adjusted to the current proof state. The actual suggestions are always presented to the user, for instance via the graphical user interface, and a command can be chosen at any time. The user can then choose the arguments for the command from all suggestions computed so far and the user can also request suggestions for a particular customized argument instantiation. As soon as the user executes a command the partial proof is updated and simultaneously the mechanism, the blackboards and the computation of their knowledge sources, are reinitialized.

Since the knowledge sources involved can be well separated into distinct societies a blackboard architecture simplifies communication. Therefore, we do not employ a different distribution model such as, for instance, a full-fledged multi-agent system. Although in the following we shall call the knowledge sources of our blackboards agents and speak thus of two layers of agent societies that communicate via blackboards the system we present should not be confused with a multi-agent system in a strong sense.

3.2 Preliminaries

The presentation in the remainder of this chapter depends mainly on a unified view of inference rules that can be applied to change a proof state and how these rules are applied by invoking an associated command. We shall therefore introduce such a unified view now and furthermore define the notion of *partial argument instantiations* of a command, which plays an important role within our architecture.

3.2.1 Inference Rules

There are several means to construct a proof in ΩMEGA both interactively and automatically. Essentially there are four major instruments for proof construction:

Rules are the implementation of the basic set of natural deduction rules as given in 2.1.3. They are the atomic components of every ΩMEGA proof; that is, in order for ΩMEGA to fully accept a proof as valid it has to be expanded fully into ND rules and successfully proof checked.

Tactics are procedures that perform abstract proof steps. They have to be expandable into subproofs containing ND rules, only. The expansion of a tactic can again contain tactics, which have to be expanded in turn.

Methods are the main component of proof planning. As described in section 2.2.4 they can be viewed as tactics plus specification. In the context of this chapter they can be treated similar to tactics.

External Reasoners are systems outside of the ΩMEGA core system. In this category are calls to automated theorem provers, computer algebra systems, or constraint solvers. Any input from an external reasoner has to be likewise expandable into natural deduction proof steps.

We will adopt a unified view on these four categories and refer to them generally as *inference rules*. Some examples of inference rules are given in figure 3.1. $\wedge_I$ is a natural deduction rule describing the introduction of a conjunction or, equivalently, the split of a conjunctive goal. $\forall^*_E$ is a simple tactic, specifying the elimination of multiple universal quantifiers. It is therefore the abbreviation for a sequence of eliminations of a universal quantifier; that is, a sequence of applications of the natural deduction rule $\forall_E$ given in 2.1.3. These two inference rules will serve as examples throughout the remainder of this chapter.

The other inference rules are $\neg_I$, $\Leftrightarrow_E$, and *Otter*. $\neg_I$ is a natural deduction rule formalizing the reasoning step that if we can derive the falsehood from A, then we know that $\neg A$ must hold. For the rule to be applicable, A must be given somewhere in the derivation of $\bot$, which is indicated by $[A]$. This also means when $\neg_I$ is applied backwards A will be introduced as new hypothesis. $\Leftrightarrow_E$ is another tactic specifying the split of an equivalence statement into two implications. Noticeable about this tactic is that it has more than a single conclusion. The latter inference rule denotes the application of an external reasoner, namely the first order automated theorem prover OTTER. It is applicable when OTTER can find a proof to justify the derivation of the conclusion C from the premises $P_1, \ldots, P_n$. In this case neither the conclusion nor any of the premises structural properties of the formulas are explicitly given. But instead the *Otter* tactic can be applied to essentially arbitrary first order formulas and, moreover, the number of premises involved in this inference rule can vary.

$$\frac{A \quad B}{A \wedge B}\,\wedge_I \qquad\qquad \frac{\forall x_1,\ldots,x_n.A}{[t_1/x_1,\ldots,t_n/x_n]A}\,\forall^*_E(t_1,\ldots,t_n)$$

$$\frac{\begin{array}{c}[A]\\ \vdots\\ \bot\end{array}}{\neg A}\,\neg_I \qquad\qquad \frac{A\Leftrightarrow B}{A\Rightarrow B \quad B\Rightarrow A}\,\Leftrightarrow_E \qquad\qquad \frac{P_1 \quad \ldots \quad P_n}{C}\,Otter$$

Figure 3.1: Examples for inference rules.

When comparing the examples in figure 3.1 we can identify five different elements any inference rule consists of: A unique name, one respective set of premises, conclusions and hypotheses, and a list of parameters. We can thus define an inference rule to be of the following general form:

$$\frac{\begin{array}{ccc}[H_1] & \cdots & [H_k]\\ & \vdots & \\ P_1 & \cdot\vdots\cdot & P_n\end{array}}{C_0 \quad \ldots \quad C_m}\,\mathcal{R}(T_1\ldots T_l)$$

This general form defines an inference rule $\mathcal{R}$ with conclusions $C_0,\ldots,C_m$, premises $P_1,\ldots,P_n$, hypotheses $H_1,\ldots,H_k$, and parameters $T_1,\ldots,T_l$. We call this the *argument pattern* of the inference rule $\mathcal{R}$ and the indexed letters its *formal arguments*. Note that the premises do not necessarily have to have all the $H_1,\ldots,H_k$ as hypotheses instead they can have any subset thereof as hypotheses. And, as the indices indicate, an inference rule has to have at least one conclusion, only, whereas all other elements are optional.

In order to apply an inference rule, at least some of its formal arguments have to be instantiated with *actual arguments*. These are either proof nodes — for the conclusions, premises and hypotheses — or arbitrary parameters, for instance, terms or lists of terms. Typically inference rules with hypotheses are applied backwards or sideways and the hypotheses do not have to be provided. A particular instantiation with actual arguments determines the effect of the application of the inference rule; that is, whether something is derived from the premises, whether new open subgoals are constructed, or whether a subproof can be successfully closed. We will refer to different effects of the application of one inference rule as different *application directions* of this inference rule. For instance, we say an inference rule is applied forwards, if new conclusions are derived from given premises and backwards, if new open subgoals are constructed in order to justify a given goal.

Let us, for instance, consider the possible application of the $\forall^*_E$ and $\wedge_I$ inference rules: $\forall^*_E$ can be applied forwards, when the premise and the list of terms is given, and to close a goal, when additionally the conclusion is given. While $\forall^*_E$ has therefore two different possible application directions, only, $\wedge_I$ has five: Forwards (A and B are given), backwards ($A \wedge B$ is given), two sideways ($A \wedge B$ and either A or B are given), and to close the subproof (all arguments are given).

3.2.2 Commands

In an interactive theorem proving environment such as ΩMEGA each inference rule has an associated *command*, which applies the tactic in a given proof state. These

$$\frac{\forall x_1,\ldots,x_n.A}{[t_1/x_1,\ldots,t_n/x_n]A}\;\forall^*_E(t_1,\ldots,t_n) \longrightarrow \frac{P}{C}\;ForallE{*}(TList)$$

$$\frac{A \quad B}{A \wedge B}\;\wedge_I \longrightarrow \frac{LConj \quad RConj}{Conj}\;AndI$$

Figure 3.2: Two inference rules and their commands

generally accept some of the actual arguments of the inference rule as input, usually in some user friendly syntax (e.g., instead of providing a proof node the user would only have to provide a unique label pointing to the proof node) and apply the associated inference rule accordingly.

Analogous to an inference rule a command has an *argument pattern*, which roughly corresponds to the argument pattern of the associated inference rule. Generally, the *formal arguments* of a command are a subset of the formal arguments of the inference rule. A command can be executed when some of its formal arguments are instantiated with *actual arguments*, which are then appropriately used to apply the associated inference rule. Taking into account that a command does not have to cater for hypotheses, we can specify a formal connection between the argument patterns of command and its associated inference rule.

$$\begin{array}{l} [H_1] \quad \cdots \quad [H_k] \\ \qquad\quad \vdots \\ \dfrac{P_1 \quad \cdots \quad P_n}{C_0 \quad \ldots \quad C_m}\;\mathcal{R}(T_1\ldots T_l) \longrightarrow \dfrac{p_{i_1}\cdots p_{i_{n'}}}{c_{j_0}\cdots c_{j_{m'}}}\;R(t_{k_1}\cdots t_{k_{l'}}) \end{array}$$

Here the argument pattern on the right corresponds to the command R that invokes $\mathcal{R}$. The command's pattern contains formal arguments for premises, conclusions and parameters that correspond to the formal arguments of the poof rule. Note that the correspondence is not necessarily one-to-one. On the one hand not all formal arguments of the inference rule must have a counterpart in the command's argument pattern, and on the other hand single arguments of the command can correspond to more than one formal argument of the inference rule. Furthermore, there are no formal arguments in the command that correspond to the hypotheses of the inference rule. Note also that all formal arguments of a command have to be uniquely named in order to be distinguishable.

As examples we examine the commands associated with the inference rules $\forall^*_E$ and $\wedge_I$ as displayed in figure 3.2. The command *AndI* is straightforward since all its formal arguments correspond directly to the formal arguments of its inference rule. In the case of *ForallE∗* only the arguments for premise and conclusion correspond directly to their counterpart in $\forall^*_E$. However, the number of parameters of $\forall^*_E$ can vary. This is modeled in the command as a single argument, *TList*, which is a term list whose length can vary.

Allowing for lists of arguments with varying length enables us to define commands with argument patterns containing a fixed number of formal arguments even when the corresponding inference rule has a non-specified number of formal arguments. This cannot only be used in the case of an undetermined number of parameters but also if the number of conclusions or premises can vary. For example, the *Otter* tactic from the last section can be associated with a command that has exactly two arguments, namely a conclusion and a list of premises.

In the remainder of this thesis when we talk about commands we always mean commands that invoke inference rules (as opposed to commands that define general functionality of the interactive system such as loading and saving of proofs etc.). We will also not always explicitly distinguish between an inference rule and its associated command unless their argument patterns vary such that a distinction is crucial for comprehension and cannot be infered from the context.

3.2.3 Partial Argument Instantiations

As we have seen earlier, in order to apply an inference rule some (not necessarily all) of its formal arguments have to be instantiated with an actual argument. This applies in turn to the command associated with the inference rule. We can formally define how formal arguments can be instantiated. For this we consider a partial proof $\mathcal{P}$ given and define the set of its proof nodes as $\mathcal{N}_{\mathcal{P}}$. Furthermore we assume a given signature Σ and a set of variables $\mathcal{V}$ for the proof such that the formulas in the proof are elements of $(\Sigma \cup \mathcal{V})^*$, the alphabet of all terms over Σ and $\mathcal{V}$. For the following definitions we shall assume that a triple $(\mathcal{P}, \mathcal{N}_{\mathcal{P}}, (\Sigma \cup \mathcal{V})^*)$ is given, consisting of a partial proof with its proof nodes, and an alphabet over a signature and a set of variables.

Definition 3.1 (Possible actual arguments): A possible actual argument is inductively defined to be either one of the following:

- an element of $\mathcal{N}_{\mathcal{P}}$,
- a term from $(\Sigma \cup \mathcal{V})^*$,
- a term position (i.e., an element of $\mathbb{N}^*$), or
- a homogeneous list of possible actual arguments.

Additionally, we define the empty actual argument to be ϵ. We define the set of all possible actual arguments $\mathcal{INST}$ as the union of all possible actual arguments and ϵ. □

The set of possible actual arguments is restricted as we have only proof nodes, terms, and positions as basic entities. This suffices for the current inference rules in ΩMEGA. However, when adding inference rules that need to be supplied with other parameters, for instance *annotations* for *annotated reasoning*, the set of possible actual arguments would have to be extended accordingly.

We now define the notion of a partial argument instantiation (PAI).

Definition 3.2 (Partial Argument Instantiation): Let C be a command for an inference rule R. Let $A_1, \ldots, A_n$ be the formal arguments of C. A partial argument instantiation PAI_C for C is a mapping from the set of formal arguments of C to the set of possible actual arguments: $PAI_C : \{A_1, \ldots, A_n\} \to \mathcal{INST} \cup \{\epsilon\}$ □

Informally, a PAI is an instantiation of formal arguments of a command with actual arguments and can thus be seen as a vector of ordered pairs consisting of formal arguments and actual arguments. Recall, that the formal arguments of a command always have distinct names and the mapping is therefore a uniquely determined assignment. We will denote PAIs as $C(A_1{:}A'_1, \ldots, A_n{:}A'_n)$ with $A'_1, \ldots, A'_n \in \mathcal{INST} \cup \{\epsilon\}$. Generally, pairs of the form $A_i : \epsilon$ can be omitted. Furthermore the command C can be omitted if it is clear from the context to which command the PAI belongs to.

Definition 3.3 (empty PAI): Let C be a command for an inference rule R. Let $A_1, \ldots, A_n$ be the formal arguments of C. We call $C(A_1{:}\epsilon, \ldots, A_n{:}\epsilon)$ (or in short $C()$) the empty PAI for C. □

Note that our definition of partial argument instantiations does not specify the actual applicability of the inference rule underlying a command with respect to the given actual arguments. The degree of instantiation of a PAI also determines the direction the corresponding inference rule is invoked in.

As examples we consider the following PAIs for the commands *ForallE∗* and *AndI*. The context is taken from a little example proof problem stated on page 47.

$$\mathit{ForallE}{*}(P{:}\, L_1.\Delta_1 \vdash \forall x_\bullet \forall y_\bullet Q(x,y), \mathit{TList}{:}(a\ b), C{:}\epsilon)$$
$$\mathit{AndI}(\mathit{Conj}{:}\, L_2.\Delta_2 \vdash Q(a,b) \wedge R, \mathit{LConj}{:}\epsilon, \mathit{RConj}{:}\, L_3.\Delta_3 \vdash R)$$

The first PAI specifies that $\forall_E^*$ is applied to the node L_1 (here given in linearized notation) containing $\forall x_\bullet \forall y_\bullet Q(x,y)$ as formula and terms a and b to eliminate the universal quantifiers. This application will eventually result in a new proof node containing $Q(a,b)$. While the PAI for *ForallE∗* specifies a forward application of the underlying inference rule, the PAI for *AndI* specifies the application of $\wedge_I$ to an open node L_2 containing $Q(a,b) \wedge R$. Since the right conjunct is already provided in the PAI, namely node L_3, the application of $\wedge_I$ will result in one new open subgoal, the one containing $Q(a,b)$.

Although PAIs contain generally proof nodes given in their linearized form we shall usually enhance readability by either only denoting the label of the proof nodes or the contained formulas. Naturally, the latter can be ambiguous because there can be several proof nodes containing the same formula. Whenever this is the case, however, we shall indicate it explicitly.

3.3 Components of the Architecture

In this section we describe the single components of the Ω-Ants-blackboard architecture. The description follows the schematic picture of the architecture given in figure 3.3. As depicted there the architecture is essentially a bridge between the central proof data structure and the user of the interactive system.

The overview of the architecture is rather informal. A formal definition of some of the components is given in section 3.6 in order to prove certain properties about the architecture.

3.3.1 Argument Agents

The bottom layer of Ω-Ants consists of societies of *argument agents*. Each society and each of its individual agents is associated with exactly one command. The goal of a single argument agent is to complete given PAIs of a command by computing additional instantiations for formal arguments not yet substituted. For this it can use some of the actual arguments already given in the PAI. The goal of a society of argument agents is to collaborate in order to compute most complete PAIs. Thereby single agents are realized as independent threads and can thus work in parallel.

Argument agents are defined with respect to three sets of formal arguments of a command:

Goal Set The formal arguments for which the argument agent tries to compute instantiations. We will also call them *goal arguments*.

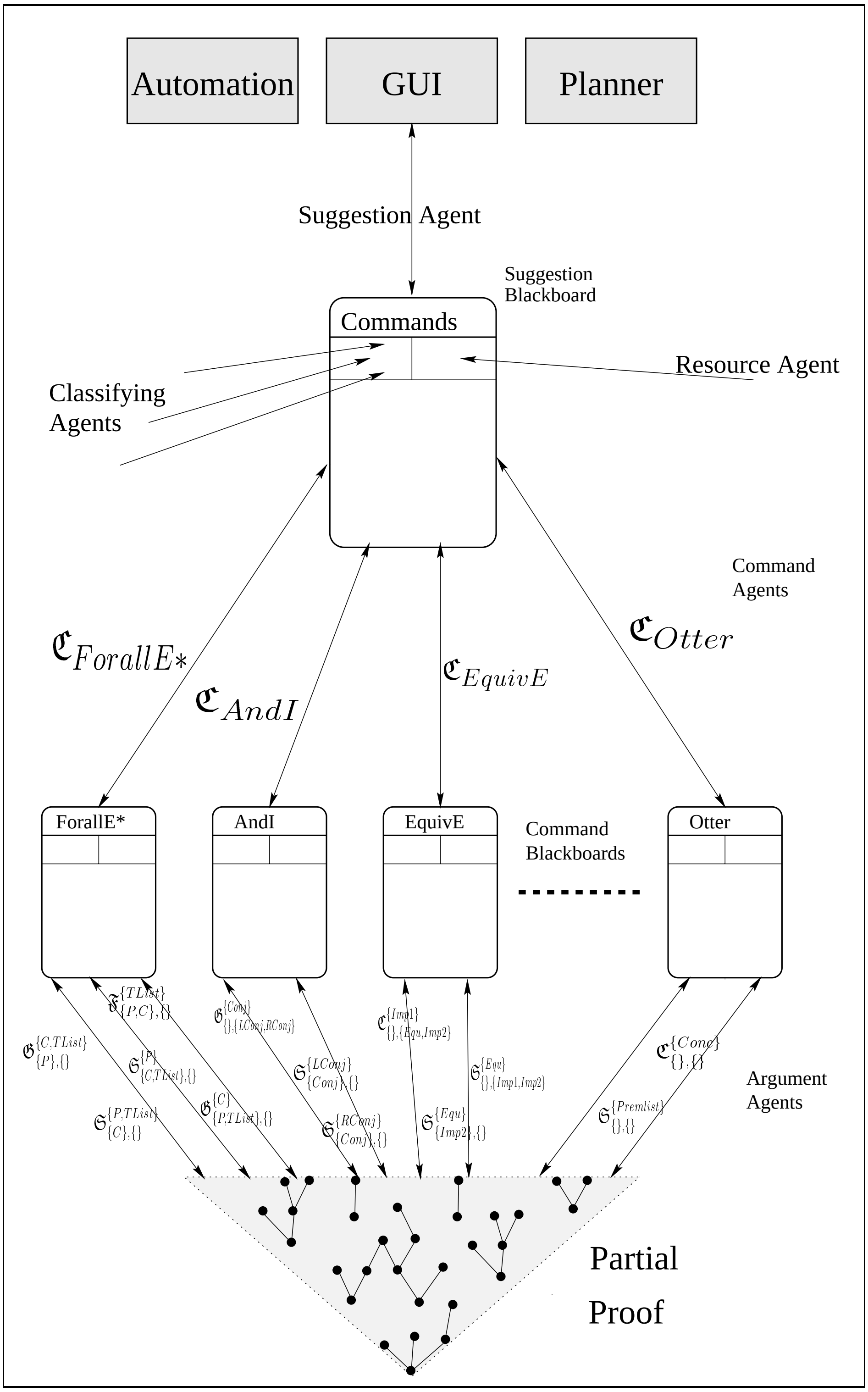

Automation
GUI
Planner
Suggestion Agent
Suggestion
Blackboard
Commands
Classifying
Agents
Resource Agent
Command
Agents
$\mathfrak{C}_{ForallE*}$
$\mathfrak{C}_{AndI}$
$\mathfrak{C}_{EquivE}$
$\mathfrak{C}_{Otter}$
ForallE*
AndI
EquivE
Command
Blackboards
Otter
$\mathfrak{F}^{\{TList\}}_{\{P,C\},\{\}}$
$\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$
$\mathfrak{C}^{\{Imp1\}}_{\{\},\{Equ,Imp2\}}$
$\mathfrak{G}^{\{C,TList\}}_{\{P\},\{\}}$
$\mathfrak{S}^{\{P\}}_{\{C,TList\},\{\}}$
$\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$
$\mathfrak{S}^{\{Equ\}}_{\{\},\{Imp1,Imp2\}}$
$\mathfrak{C}^{\{Conc\}}_{\{\},\{\}}$
Argument
Agents
$\mathfrak{S}^{\{P,TList\}}_{\{C\},\{\}}$
$\mathfrak{G}^{\{C\}}_{\{P,TList\},\{\}}$
$\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$
$\mathfrak{S}^{\{Equ\}}_{\{Imp2\},\{\}}$
$\mathfrak{S}^{\{Premlist\}}_{\{\},\{\}}$
Partial
Proof

Figure 3.3: The Ω-Ants blackboard architecture.

$$\begin{aligned}
\mathfrak{S}^{\{P\}}_{\{\},\{C,TList\}} &= \{P\text{: } P \text{ is a universal quantification}\}\\
\mathfrak{S}^{\{P,TList\}}_{\{C\},\{\}} &= \{P\text{: One scope of } P \text{ matches } C\text{, } TList\text{: Matching terms in the right order}\}\\
\mathfrak{S}^{\{P\}}_{\{C,TList\},\{\}} &= \{P\text{: After instantiating the quantifiers of } P \text{ wrt. } TList \text{ the resulting term}\\
&\qquad \text{is identical to } C\}\\
\mathfrak{G}^{\{C,TList\}}_{\{P\},\{\}} &= \{C\text{: } C \text{ matches one scope of } P\text{, } TList\text{: The matching terms}\}\\
\mathfrak{G}^{\{C\}}_{\{P,TList\},\{\}} &= \{C\text{: } C \text{ matches one scope of } P \text{ wrt. the terms of } TList\}\\
\mathfrak{F}^{\{TList\}}_{\{P,C\},\{\}} &= \{TList\text{: The matching terms of } C \text{ and one scope of } P\}
\end{aligned}$$

Table 3.1: Argument agents of the *ForallE∗* command.

Dependency Set Formal arguments that have to be instantiated with actual arguments in a given PAI for the computations of the argument agent to be successful. We will refer to the arguments of the dependency set also as *necessary arguments*.

Exclusion Set Formal arguments that must not already be instantiated in a PAI for the computations of the argument agent to be meaningful. These will also be referred to as *disturbing arguments*.

Argument agents are specified via *argument predicates* and *functions*, which model the dependencies between different formal arguments of a command. The difference between argument predicates and functions is that the former contain conditions the formula of a proof node has to fulfill and which can be used for search, whereas the latter contain algorithms to compute additional arguments with respect to some already given arguments. Consequently, argument agents can be divided into *predicate* and *function agents* where the former search in the proof tree and the latter conduct computations. In fact, predicate agents can be further subdivided into *goal* and *support agents* depending on whether they search in the open nodes or the support nodes of a proof. This distinction will be elaborated in section 3.3.6.

Since the distinction between the different types of agents is with respect to their goal we have to clarify the case of an agent that has more than one goal argument. In this case the order of the goal arguments is important. The first of these, called the *primary goal argument*, is the argument the agent primarily computes for. All subsequent elements of the goal set are arguments, which can be instantiated additionally if the agent is successful. Thus, unlike dependency and exclusion set, the goal set should rather be seen as an ordered tuple where the first element determines the type of the argument agent.

We will denote goal, support, and function agents with $\mathfrak{G}$, $\mathfrak{S}$, and $\mathfrak{F}$, respectively. The goal set will be attached as superscript and dependency and exclude set as indices in this order. For instance, $\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}}$ denotes a goal agent for the *AndI* command whose aim is to compute an instantiation for the formal argument *Conj* in a PAI where *LConj* is already instantiated and *RConj* must not yet be present. The full set of argument agents for the *ForallE∗* and *AndI* commands are given in tables 3.1 and 3.2, respectively. The predicates and functions are given very informally in plain text, for a formal definition we refer to section 3.6.

$$\begin{aligned}
\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}} &= \big\{Conj\colon\ Conj \text{ is a conjunction}\big\}\\
\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}} &= \big\{Conj\colon \text{ Left conjunct of } Conj \text{ is equal to } LConj\big\}\\
\mathfrak{G}^{\{Conj\}}_{\{RConj\},\{LConj\}} &= \big\{Conj\colon \text{ Right conjunct of } Conj \text{ is equal to } RConj\big\}\\
\mathfrak{G}^{\{Conj\}}_{\{LConj,RConj\},\{\}} &= \big\{Conj\colon \text{ Left conjunct of } Conj \text{ is equal to } LConj \text{ and the}\\
&\qquad\qquad \text{right conjunct of } Conj \text{ is equal to } RConj\big\}\\
\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}} &= \big\{LConj\colon\ LConj \text{ is equal to the left conjunct of } Conj\big\}\\
\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}} &= \big\{RConj\colon\ RConj \text{ is equal to the right conjunct of } Conj\big\}
\end{aligned}$$

Table 3.2: Argument agents of the *AndI* command.

We examine some of the agents in more detail. $\mathfrak{S}^{\{P\}}_{\{\},\{C,TList\}}$ in table 3.1 specifies that the required argument has to be a proof node that contains a universally quantified formula. The agent's computation does not depend on any necessary arguments in the PAI it can be applied to. Moreover, the respective PAI must not yet have instantiations for the C and $TList$ arguments. This exclusion set is necessary since otherwise the agent might wrongly complete an already partially instantiated PAI (e.g., where an instantiation for C is already given). This could lead to non-applicable PAIs, because the predicate of the agent was under-specified with respect to the available information in this case. Note that the two occurrences of P in the agent's predicate have two different denotations: The first denotes the name of the formal argument of the command an actual argument is assigned to, whereas the second denotes an actual proof node that is tested as possible actual argument.

The next agent $\mathfrak{S}^{\{P,TList\}}_{\{C\},\{\}}$ has more than one element in its goal set. Its primary goal argument determines it as a goal agent. Its goal is to find an instantiation for P, which is a universally quantified formula whose scope matches the formula of the instantiation for C. Since $\forall^*_E$ can eliminate several quantifiers the predicate of $\mathfrak{S}^{\{P,TList\}}_{\{C\},\{\}}$ checks whether any of the scopes of a possible instantiation for P can be matched. In case the match is successful we automatically get the proper matcher whose domain is used to determine subsequently the $TList$ argument.

The $\mathfrak{S}^{\{P\}}_{\{C,TList\},\{\}}$ agent performs the same matching process, however, with $TList$ already instantiated. Thus, C does not only have to match one scope of P but also the domain of the matcher has to be equivalent to $TList$. Here the idea is, that the more information is available the more this information should be taken into account by a more specialized agent. The following two agents $\mathfrak{G}^{\{C,TList\}}_{\{P\},\{\}}$ and $\mathfrak{G}^{\{C\}}_{\{P,TList\},\{\}}$ are analogous to the previous two with the exception that since they are support agents they look for a possible instantiation for C with respect to P. Finally, $\mathfrak{F}^{\{TList\}}_{\{P,C\},\{\}}$ is the only function agent in this society of argument agents. Its goal is to compute the proper value of the $TList$ when given instantiations for the P and C arguments.

The agents for the *AndI* command are rather simple and we will not explain them in detail. The only slightly exceptional agents are the last two: $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ and $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$. So far, all discussed agents contained each of the formal arguments of the respective command either in the goal, dependency, or exclusion set. Here, however, is a slight difference: The former agent does not contain *RConj* and the latter not *LConj*. This is due to the fact that both formal arguments *RConj* and *LConj* are independent from each other and each only depends on the *Conj* argument. For instance, can $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ agent complete a PAI in which both *Conj* and *RConj* are already instantiated without computing a non-applicable PAI.

The following are two examples for computations of argument agents. The context is again taken from the example proof problem on page 47.

$$\mathit{ForallE}{*}(P{:}\forall x_{\blacksquare}\forall y_{\blacksquare}Q(x,y)) \xrightarrow{\mathfrak{S}^{\{C,TList\}}_{\{P\},\{\}}} \mathit{ForallE}{*}(P{:}\forall x_{\blacksquare}\forall y_{\blacksquare}Q(x,y), \mathit{TList}{:}(a\ b), C{:}Q(a,b))$$

$$\mathit{AndI}(\mathit{Conj}{:}Q(a,b) \wedge R) \xrightarrow{\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}} \mathit{AndI}(\mathit{Conj}{:}Q(a,b) \wedge R, \mathit{RConj}{:}R)$$

The upper is an example for the computation of the $\mathfrak{S}^{\{C,TList\}}_{\{P\},\{\}}$ agent belonging to the *ForallE*∗, which applied to a PAI in which P is already instantiated, returns instantiations for C and $TList$. The lower gives the computation of the $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$ agent of the *AndI* command, which returns an instantiation for the *RConj* argument when applied to a PAI in which at least *Conj* is instantiated.

The single agents of each society of argument agents are not necessarily fixed; that is, we can both vary the number of agents as well as the agent definition itself. In fact, it is possible to add, delete or change argument agents in Ω-ANTS even at runtime.

3.3.2 Command Blackboards

So far we have only examined how single argument agents can complete already given PAIs. But as already mentioned the goal of a whole society of argument agents is to collaborate in order to compute the most complete PAIs possible. To collaborate the single agents need a means to exchange results. This is achieved with the help of a blackboard called the *command blackboard*. Command blackboards form the second layer in our architecture as presented in figure 3.3.

Each society of argument agents has one associated command blackboard, which in turn is associated with the same command the argument agents belong to. Entries on the blackboard consist of single PAIs together with the information whether an agent has already read it. The communication via the blackboard works as follows: An agent examines all PAIs it has not previously visited in order to determine those it can apply its argument predicate or function to; that is, the agent checks for each new PAI whether all its necessary arguments and none of its goal and disturbing arguments are contained. All the checked blackboard entries are marked as read by the agent ensuring that they will not again be considered by the same agent.

Once the agent has found all triggering PAIs out of the set of new entries, it executes its computations for each: It either performs a search in the current partial proof with respect to its predicate or executes its function to compute an actual argument. In case a computation was successful its result is used to create a new PAI, which consists of the old PAI augmented by the newly computed instantiations. The agent's computation can also return multiple results for one formal argument, for example, several lines satisfying the predicate, which will result in several new PAIs to be written on the blackboard simultaneously.

Since argument agents only read entries and write new enlarged copies on the blackboard leaving the original entry unchanged there is no need for conflict resolution between agents. Conflict resolution can become necessary in blackboard architectures where knowledge sources working in parallel try to work with the

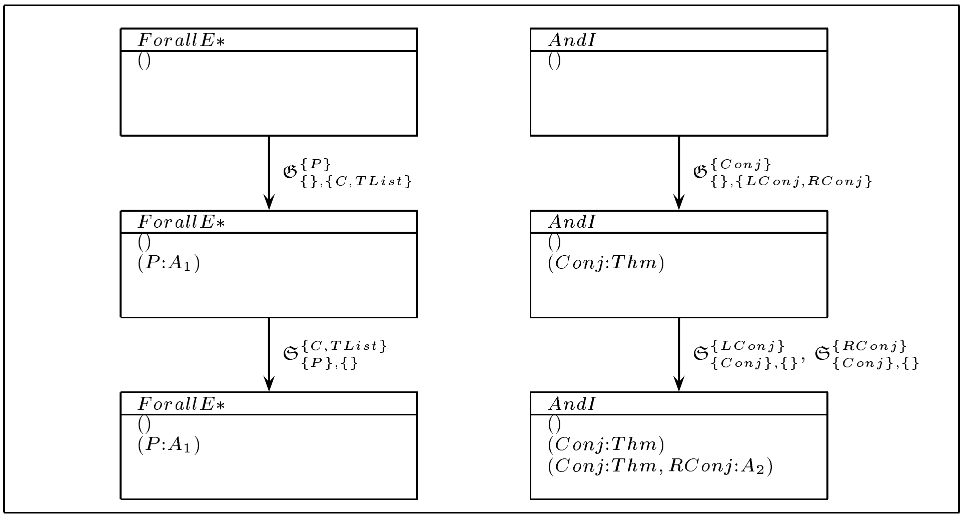

$ForallE*$
$()$
$\mathfrak{G}^{\{P\}}_{\{\},\{C,TList\}}$
$ForallE*$
$()$
$(P{:}A_1)$
$\mathfrak{S}^{\{C,TList\}}_{\{P\},\{\}}$
$ForallE*$
$()$
$(P{:}A_1)$
$AndI$
$()$
$\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$
$AndI$
$()$
$(Conj{:}Thm)$
$\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}, \mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$
$AndI$
$()$
$(Conj{:}Thm)$
$(Conj{:}Thm, RConj{:}A_2)$

Figure 3.4: Communication on command blackboards.

same entry and thereby changing it. Then changes of one knowledge source can render changes of another knowledge source obsolete or even make them impossible. This case, however, cannot arise in our context, because even if two agents start working with the same entry at the same time, any results will be added as new extended entries. These new entries can then be inspected by the respective other agents. Likewise can the original entry still be read and used by other agents. The concept also permits different possible argument instantiations since a less complete PAI may nevertheless be a valid instantiation for the argument of a command.

In order for any agent to be triggered the blackboard needs to be initialized. This is normally done by writing the empty PAI on the blackboard, which triggers agents with an empty dependency set to perform their computations. If those agents return any useful results, they write these on the blackboard, thereby possibly triggering other agents. Once the proof state changes, for instance, when the user executes a suggested command, each command blackboard is reinitialized starting a new cycle of agent computations.

Blackboards can also be initialized with any other possible PAI. This is done for instance when the user performs a specific query by already supplying some instantiations for actual arguments and asking for a completion with respect to these arguments. Then the argument agents commence their computations with respect to this PAI instead of the empty one.

We demonstrate the collaboration and communication within the two argument agent societies from the preceding section with the example of the following trivial proof.

$$\begin{array}{llll}
A_1. & {\scriptstyle A_1} \vdash \forall x_\bullet \forall y_\bullet Q(x,y) & & \text{(Hyp)} \\
A_2. & {\scriptstyle A_2} \vdash R & & \text{(Hyp)} \\
 & \quad\vdots & & \\
Thm. & {\scriptstyle A_1,A_2} \vdash Q(a,b) \wedge R & & \text{(Open)}
\end{array}$$

Note that the proof is presented in the linearized ND calculus, as introduced in chapter 2.1.3; that is, the proof nodes have been denoted as uniquely labeled lines.

The communication between the single agents evolves as shown in figure 3.4. The topmost row represents the two blackboards involved after initialization. We assume that the two command blackboards are initialized with the empty PAI each. This triggers the $\mathfrak{S}^{\{P\}}_{\{\},\{C,TList\}}$ and $\mathfrak{S}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$ of the $ForallE*$ and *AndI* commands, respectively, since these are the only two agents not depending on necessary arguments. The respective agents triggered by entries on the blackboard are given as captions of the downward arrows. The computation of each of the two agents eventually gives rise to a new entry on each blackboard, namely $(P{:}A_1)$ and $(Conj{:}Thm)$, which in turn trigger the next agents to run. Since after the first step nothing more is going to happen on the $ForallE*$ blackboard we concentrate on the *AndI* blackboard, instead. We can observe that the $(Conj{:}Thm)$ entry triggers two agents to run, namely $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ and $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$. The two agents simultaneously start their search but only the latter produces a result, line A_1, which contains the right conjunct of the formula in line Thm. This leads to the third blackboard entry $(Conj{:}Thm, RConj{:}A_2)$.

Figure 3.4 can give the impression that the communication on the blackboard obeys a certain cycle when entries are added to all command blackboards, this is however wrong in practice. Agents starting their computations in parallel at the same time will not always produce a result at the same time as well. Instead, this depends on how long the agents' computation take and how the agents' threads are scheduled by the programming language and/or operating system.

The search space spanned by the agents' computation is essentially a tree structure, where the vertices are the different PAIs and the edges are labeled with the agents. The root vertex of the tree is the empty PAI. In each proof state the tree structure contains all possible PAIs with most complete PAIs contained in the leafs. The branching points then correspond either to concurrent computations if two edges are labeled with different agents or, for edges labeled with the same agent, different possible instantiations, which are subject to the order of the agent's search. However, the search behavior is not a simple breadth first search, instead the traversal of the tree depends on how fast the single agents' computations are and how their threads are scheduled. This also determines the order of the PAIs added to the blackboard. Moreover, not all vertices of the tree correspond to entries of the blackboard since we do not allow for double entries, but there can be repeated vertices. This gives the computation in Ω-Ants an *anytime behavior*; that is, the more time the agents have for their computation the farther they can traverse the search space and the better the suggestions on the blackboard become.

3.3.3 Command Agents

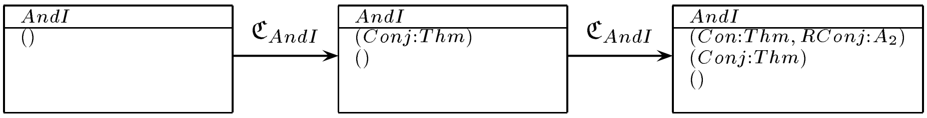
So far we have seen how argument agents can communicate via the command blackboard by reading existing entries and writing new augmented copies to the blackboard. This is done in no particular order although it is desirable to have the entries sorted according to certain criteria, for instance, with respect to the number of actual arguments instantiated in the single PAIs. Unfortunately, this order is not automatically achieved since due to the indeterminism mentioned in the preceding section some agents might still add PAIs while other agents have already completed entries added earlier by the same agent.

The sorting of entries on a blackboard is done by a *command agent*. Command agents form the next layer of our architecture depicted in figure 3.3. Each command agent is associated with one command that surveys the associated command blackboard. Its task is to constantly monitor the blackboard, and as soon as a new entry is added it sorts the prolonged list of PAIs according to given heuristic criteria. The command agent also reinitializes the blackboard either when Ω-Ants is reset when a command has been executed, or if the user executes a particular query.

Command agents can be equipped with different sorting criteria depending on the way Ω-Ants is employed. However, these criteria can be changed, even at runtime and any changes take effect with the next reset of Ω-Ants. Generally, the heuristics are common to all command agents but can, if desired, also be changed in individual agents. There exists a set of standard heuristics Ω-Ants is usually initialized with and whose main criterion is to prefer the most complete PAI on the blackboard. Among those with the same degree of instantiation the system can use further criteria such as which PAI contains the most recently derived proof nodes. This criterion is presented in more detail in section 3.3.6.3.

Additionally, each command agent can be equipped with an applicability test for PAIs. A PAI is then considered as applicable only if certain sets of formal arguments are actually instantiated. For instance, for the *AndI* command PAIs containing actual arguments for at least the *Conj* or both *RConj* and *LConj* argument are considered applicable, whereas PAIs containing only instantiations of *RConj* or *LConj* are not.

Command agents are denoted by $\mathfrak{C}$ and a single index indicating the command it is associated with. Thus, in our previous example the command agents monitoring the *ForallE∗* and *AndI* blackboards are $\mathfrak{C}_{ForallE*}$ and $\mathfrak{C}_{AndI}$. Given below is the *AndI* command blackboard sorted by the latter agent, assuming that it uses the standard heuristics, where the most complete PAI is sorted to the top. Hence, in this simple example the last computed PAI is always sorted on top.



3.3.4 The Suggestion Blackboard

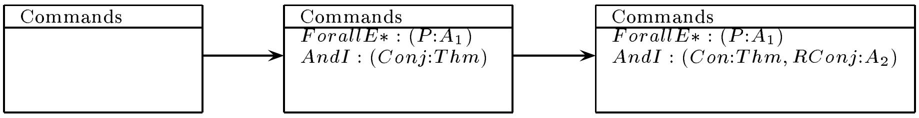
As seen in the last section each command agent constantly monitors its respective command blackboard and if necessary sorts its entries. Therefore, it also has knowledge about which of the given PAIs on the blackboard is the heuristically preferred entry. This information is passed to the next level of our architecture, to the ***suggestion blackboard*** (see figure 3.3). The suggestion blackboard gathers entries consisting of commands applicable in a particular proof state together with their respective preferred PAIs.

In detail, the suggestion blackboard is filled as follows: As soon as a command agent has an applicable PAI as best suggestion on its command blackboard it constructs a suggestion blackboard entry consisting of the command's name and the respective PAI and writes this entry on the blackboard. Whenever the same command agent detects a new best PAI on its command blackboard it then updates its entry on the suggestion blackboard by replacing the old with the new PAI. Here the applicability of the PAI is decided using the command agent's applicability test described in the preceding section. In case a command agent is not equipped with an explicit applicability test, each non-empty PAI is considered to be applicable and hence is propagated.

The structure of the blackboard entries does not mean that only the heuristically best entries of each command blackboard can be further processed. In fact, when the user chooses interactively one of the suggested commands, it is possible to choose again from the PAIs computed for this command so far.

The suggestion blackboard is initially empty. It is reinitialized whenever Ω-Ants is reset, for instance if a command has been executed.

We observe how the suggestion blackboard is updated in the case of our example below. At the beginning the suggestion blackboard is empty since all command blackboards contain only the empty PAI. The blackboard in the middle then shows the situation after the first suggestions have been produced by the argument agents and propagated by the command agents. Here we already have two possible applicable commands. Then a more complete PAI is computed for the *AndI* command leading to an update of the entry on the suggestion blackboard as shown on the right.



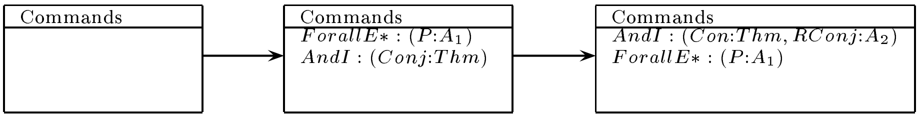
3.3.5 The Suggestion Agent

Similar to the command blackboards the entries on the suggestion blackboard are a priori not sorted. Thus, we introduce an equivalent to the command agents for the suggestion blackboard, the *suggestion agent*. Its role is to constantly monitor the suggestion blackboard and as soon as a new entry is added or an old one is updated, this agent sorts the entries on the board. The sorting is again performed with respect to heuristics that can also be subject to change (see 3.4 for details).

One of the standard heuristics used is, for instance, to prefer suggestions that have on average the most complete PAI; that is, we use the ratio of the number of formal arguments of a command and the number of actual arguments the PAI contains. This, for example, always prefers commands that have all formal arguments instantiated which is motivated by the hope that a command that can be supplied with all possible actual arguments might invoke its inference rule to close an open subproblem.

Another task of the suggestion agent is to suitably process the entries of the suggestion blackboard. For instance the entries are displayed to the user on a board of applicable commands in a graphical user interface. The entries can be also passed to other algorithms for further use. An example for this is the automation of the command application as elaborated in section 3.5. In practice the suggestion agent contains a function specifying what to do with the entries on the blackboard. This function is executed whenever a new entry is added to the suggestion blackboard or the old entries are resorted.

As an example consider again the update sequence of the suggestion blackboard from the preceding section. Considering that we use the standard heuristic to prefer the entries with respect to the ratio of instantiation of their PAIs the entries on the very right blackboard are actually reordered. This is because both *AndI* and *ForallE∗* have the same number of formal arguments, however the attached PAI of *AndI* has more actual arguments given those of *ForallE∗*.



Both the processing function and the sorting heuristics of the suggestion agent can be changed depending on the intended use of the suggestions. For instance, other heuristical criteria for sorting the command suggestions can include:

- Inference rules performing backward reasoning are preferred to those for forward reasoning in order to achieve a more goal directed reasoning process.

- Tactics are preferred to rules since they might make larger steps in the proof.
- Suggestions are sorted such that suggestions introducing the least new open subgoals are preferred to those introducing the most in order to reduce branching in the proof.
- Inference rules that are applicable in every proof step, as for instance proof by contradiction, are always sorted to the end of the list of suggestions to avoid their redundant application.

3.3.6 Foci

So far the argument agents search blindly either among all open nodes or among all support nodes of the proof. The fact that the logical structure of the proof in question is not taken into account can lead to major errors in the suggested arguments, as a set of arguments might be proposed that is logically incompatible. This can lead not only to non-applicable entries on the blackboard but also to an unnecessary amount of search. Therefore, we elaborate in this section a focus technique that suitably guides the search for default suggestions by restricting it to certain subproblems. Thereby we exploit the property of natural deduction proofs that have a strong, intrinsic logical dependencies between the single proof nodes. In other calculi, for instance in resolution calculus, these dependencies are generally much weaker. The focus mechanism keeps explicitly track of this implicitly given structural information and also of the chronological information given in a partial proof and enables the reuse of this information in case some already justified proof nodes are deleted and the system backtracks to a previous proof state.

3.3.6.1 Definitions

We shall first give the definitions of the main concepts involved and then consider an example. For the following definitions let $\mathcal{P}$ be a partial proof with its set of proof nodes $\mathcal{N}_{\mathcal{P}}$.

Definition 3.4 (Chronological node order): We define a total order $<_n : \mathcal{N}_{\mathcal{P}} \times \mathcal{N}_{\mathcal{P}}$ on the proof nodes of $\mathcal{P}$ such that for all $n_1, n_2 \in \mathcal{N}_{\mathcal{P}}$ holds: $n_1 <_n n_2$, iff proof node n_1 was inserted in $\mathcal{P}$ before n_2. We call $<_n$ the *chronological node order of* $\mathcal{P}$. □

In practice, each proof node gets assigned a non-negative integer when it is introduced into the proof. This is done by having a *node counter* that is incremented whenever a new node has been introduced into the proof. The theorem has an initial value of 0, its original assumptions are then successively numbered in no particular order. In case an application of an inference rule introduces more than one new node at a time, these are likewise incrementally numbered although they are essentially introduced simultaneously. This way we can ensure that $<_n$ remains a total order.

Definition 3.5 (Focus): Let $n \in \mathcal{N}_{\mathcal{P}}$ and $\mathcal{SN}, \mathcal{DN} \subseteq \mathcal{N}_{\mathcal{P}}$. We call the triple $f = (\mathcal{SN}, n, \mathcal{DN})$ a *focus*, if $\mathcal{SN}$ is a set of *support nodes* of n (i.e., nodes that can be used to derive n) and $\mathcal{DN}$ a set of *descendant nodes* of n (i.e., nodes that have been derived from n). n is called the *focused node* of f. If n is an open node, we call f *open*, otherwise *closed*. The set of all foci of $\mathcal{P}$ is denoted by $F_{\mathcal{P}}$. □

Note that both $\mathcal{SN}$ and $\mathcal{DN}$ can contain open nodes.

Definition 3.6 (Focus context): Using foci as base constructions we inductively define the set $FC_{\mathcal{P}}$ of *focus contexts* of $\mathcal{P}$ as the smallest set containing:

(i) $F_{\mathcal{P}} \subseteq FC_{\mathcal{P}}$

(ii) Let $fc_1, \ldots, fc_k \in FC_{\mathcal{P}}$ be a set of focus contexts with respective sets of derived nodes $\mathcal{DN}_1 \ldots \mathcal{DN}_k$. Let $n \in \mathcal{N}_{\mathcal{P}}$ with premises $n_1, \ldots, n_k$, where $n_i \in \mathcal{DN}_i$ for $1 \leq i \leq k$, and let $\mathcal{DN}$ be the set of descendant nodes of n. Then the triple $fc = ((fc_1, \ldots, fc_k), n, \mathcal{DN})$ is called a *focus context* of $\mathcal{P}$ with $fc \in FC_{\mathcal{P}}$. □

Definition 3.7 (Foci priority order): Given a set $S \subseteq FC_{\mathcal{P}}$ of focus contexts of $\mathcal{P}$. A total ordering $\prec$: $S \times S$ is called a *foci priority order* on S. □

In practice, we proceed for the foci priority order analogously to the chronological node order. Foci are assigned a non-negative integer value, according to a *foci counter*. This counter is incremented whenever a new focus is introduced.

Definition 3.8 (Proof context): Let $S \subseteq FC_{\mathcal{P}}$ be a set of focus contexts of $\mathcal{P}$, $<_n$ a chronological node order for $\mathcal{P}$ and $\prec$ a foci priority order on S. We then call the triple $pc = (S, <_n, \prec)$ a *proof context* for $\mathcal{P}$. Note that for each focus context fc in S the restriction of $<_n$ on the set of support nodes of fc is unique. □

Definition 3.9 (Active focus): Given a proof context $pc = (S, <_n, \prec)$. Then we call the uniquely defined open focus context $fc \in S$ that is maximal with respect to $\prec$ the *active focus context* of pc. □

The active focus essentially constitutes the subproblem currently under consideration. For interaction a user can naturally explicitly change the focused subproblem to any other open node. Then the open focus context for this node is promoted to become the new active focus by assigning it the next value of the foci counter.

Initially a partial proof consists of a proof context containing exactly one focus context. Application and retraction of different inference rules give rise to various changes of focus contexts and transitions of the proof context while constructing a proof. The following definition lists in more detail the different types of transitions of proof contexts with respect to different possible inference rules.

Definition 3.10 (Transition of proof contexts): Let $\mathcal{P}$ be a partial proof with proof context $pc = (S, <_n, \prec)$ with $S = \{fc_1, \ldots, fc_l\}$, where the fc_i are focus contexts of $\mathcal{P}$. Let $\mathcal{R}$ be an inference rule. We define a *transition of the poof context pc* as the proof context $pc' = (S', <_n', \prec')$, where the set $S' = \{fc'_1, \ldots, fc'_{l'}\}$ consists of the focus contexts of pc after the application of $\mathcal{R}$ to elements of $\mathcal{N}_{\mathcal{P}}$ and $<_n', \prec'$ are the respective extended orderings. Thus, the transformation of the single focus contexts determines the transition of pc. Depending on the form of $\mathcal{R}$ and its application direction we can identify the following transformations for all $fc \in S$, where $fc = (\mathcal{SN}, n, \mathcal{DN})$:

1. Suppose the rule $\mathcal{R}$ is of the form $\dfrac{P}{C}\ \mathcal{R}$

   (a) Let $P \in \mathcal{N}_{\mathcal{P}}$ (forward application):

$$fc' = \begin{cases} (\mathcal{SN} \cup \{C\}, n, \mathcal{DN}) & \text{if} \quad P \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \{C\}) & \text{if} \quad P \in \mathcal{DN} \text{ or } n = P \\ fc & \text{otherwise} \end{cases}$$

(b) Let $C \in \mathcal{N}_{\mathcal{P}}$ (backward application):

$$fc' = \begin{cases} (\mathcal{SN}, P, \mathcal{DN} \cup \{C\}) & \text{if} \quad n = C \\ (\mathcal{SN} \cup \{P\}, n, \mathcal{DN}) & \text{if} \quad C \in \mathcal{SN} \\ fc & \text{otherwise} \end{cases}$$

Note that the case $C \in \mathcal{DN}$ is not possible since C is an open node and thus not derived from any element of $\mathcal{N}_{\mathcal{P}}$.

(c) Let $P, C \in \mathcal{N}_{\mathcal{P}}$ (closing application) and let $\widehat{fc} = (\widehat{\mathcal{SN}}, C, \widehat{\mathcal{DN}})$:

$$fc' = \begin{cases} fc \text{ and closed} & \text{if} \quad fc = \widehat{fc} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}} \cup \{C\}) & \text{if} \quad P \in \mathcal{DN} \text{ or } n = P \\ fc & \text{otherwise} \end{cases}$$

2. Suppose $\mathcal{R}$ is of the form $\dfrac{\begin{array}{c}[H]\\ \vdots \\ P\end{array}}{C}\ \mathcal{R}$

(a) Let $P, H \in \mathcal{N}_{\mathcal{P}}$ (forwards): Similar to case 1a.

(b) Let $C \in \mathcal{N}_{\mathcal{P}}$ (backwards):

$$fc' = \begin{cases} (\mathcal{SN} \cup \{H\}, P, \mathcal{DN} \cup \{C\}) & \text{if} \quad n = C \\ (\mathcal{SN} \cup \{P, H\}, n, \mathcal{DN}) & \text{if} \quad C \in \mathcal{SN} \\ fc & \text{otherwise} \end{cases}$$

(c) Let $P, H, C \in \mathcal{N}_{\mathcal{P}}$ (closing): Similar to case 1c.

3. Suppose $\mathcal{R}$ is of the form $\dfrac{P_1 \quad P_2}{C}\ \mathcal{R}$

(a) Let $P_1, P_2 \in \mathcal{N}_{\mathcal{P}}$ (forwards):

$$fc' = \begin{cases} (\mathcal{SN} \cup \{C\}, n, \mathcal{DN}) & \text{if} \quad P_1, P_2 \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{DN} \text{ or } P_2 \in \mathcal{DN} \text{ or} \\ & \qquad n = P_1 \text{ or } n = P_2 \\ fc & \text{otherwise} \end{cases}$$

(b) Let $C \in \mathcal{N}_{\mathcal{P}}$ (backwards):

$$fc' = \begin{cases} \begin{cases} fc'_1 = (\mathcal{SN}, P_1, \mathcal{DN} \cup \{C\}) \\ fc'_2 = (\mathcal{SN}, P_2, \mathcal{DN} \cup \{C\}) \end{cases} & \text{if} \quad n = C \\ (\mathcal{SN} \cup \{P_1, P_2\}, n, \mathcal{DN}) & \text{if} \quad C \in \mathcal{SN} \\ fc & \text{otherwise} \end{cases}$$

Note that in case $n = C$ we have a split of focus contexts. This is equivalent to two new foci being added to S.

(c) Let $P_1, C \in \mathcal{N}_{\mathcal{P}}$ (sideways right) and let $\widehat{fc} = (\widehat{\mathcal{SN}}, C, \widehat{\mathcal{DN}})$:

$$fc' = \begin{cases} (\mathcal{SN}, P_2, \mathcal{DN} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{SN} \text{ and } n = C \\ (\mathcal{SN} \cup \{P_2\}, n, \mathcal{DN}) & \text{if} \quad C, P_1 \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{DN} \text{ or } n = P_1 \\ fc & \text{otherwise} \end{cases}$$

Observe, that in the case $n = C$ we have $\widehat{\mathcal{DN}} \subseteq \mathcal{DN}$, necessarily.

(d) Let $P_2, C \in \mathcal{N}_{\mathcal{P}}$ (sideways left): Symmetrical to case 3c.

(e) Let $P_1, P_2, C \in \mathcal{N}_{\mathcal{P}}$ (closing) and let $\widehat{fc} = (\widehat{\mathcal{SN}}, C, \widehat{\mathcal{DN}})$:

$$fc' = \begin{cases} fc \text{ and closed} & \text{if} \quad fc = \widehat{fc} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{DN} \text{ or } P_2 \in \mathcal{DN} \\ & \quad\;\; \text{or } n = P_1 \text{ or } n = P_2 \\ fc & \text{otherwise} \end{cases}$$

4. Suppose $\mathcal{R}$ is of the form $\dfrac{P_1 \quad \begin{array}{c}[H] \\ \vdots \\ P_2\end{array}}{C}\,\mathcal{R}$

(a) Let $H, P_1, P_2 \in \mathcal{N}_{\mathcal{P}}$ (forwards): Similar to case 3a.

(b) Let $C \in \mathcal{N}_{\mathcal{P}}$ (backwards):

$$fc' = \begin{cases} \begin{cases} fc'_1 = (\mathcal{SN}, P_1, \mathcal{DN} \cup \{C\}) \\ fc'_2 = (\mathcal{SN} \cup \{H\}, P_2, \mathcal{DN} \cup \{C\}) \end{cases} & \text{if} \quad n = C \\ (\mathcal{SN} \cup \{P_1, P_2, H\}, n, \mathcal{DN}) & \text{if} \quad C \in \mathcal{SN} \\ fc & \text{otherwise} \end{cases}$$

Note that in case $n = C$ we have again a split of focus contexts.

(c) Let $P_1, C \in \mathcal{N}_{\mathcal{P}}$ (sideways right) and let $\widehat{fc} = (\widehat{\mathcal{SN}}, C, \widehat{\mathcal{DN}})$:

$$fc' = \begin{cases} (\mathcal{SN} \cup \{H\}, P_2, \mathcal{DN} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{SN} \text{ and } n = C \\ (\mathcal{SN} \cup \{P_2, H\}, n, \mathcal{DN}) & \text{if} \quad C, P_1 \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}} \cup \{C\}) & \text{if} \quad P_1 \in \mathcal{DN} \text{ or } n = P_1 \\ fc & \text{otherwise} \end{cases}$$

(d) Let $H, P_2, C \in \mathcal{N}_{\mathcal{P}}$ (sideways left): Similar to case 3d.

(e) Let $H, P_1, P_2, C \in \mathcal{N}_{\mathcal{P}}$ (closing): Similar to case 3e.

5. Suppose $\mathcal{R}$ is of the form $\dfrac{P}{C_1 \quad C_2}\,\mathcal{R}$

(a) Let $P \in \mathcal{N}_{\mathcal{P}}$ (forwards):

$$fc' = \begin{cases} (\mathcal{SN} \cup \{C_1, C_2\}, n, \mathcal{DN}) & \text{if} \quad P \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \{C_1, C_2\}) & \text{if} \quad P \in \mathcal{DN} \text{ or } n = P \\ fc & \text{otherwise} \end{cases}$$

(b) Let $C_1, C_2 \in \mathcal{N}_{\mathcal{P}}$ (backwards) and let $\widehat{fc_1} = (\widehat{\mathcal{SN}_1}, C_1, \widehat{\mathcal{DN}_1})$, $\widehat{fc_2} = (\widehat{\mathcal{SN}_2}, C_2, \widehat{\mathcal{DN}_2})$:

$$fc' = \begin{cases} (\widehat{\mathcal{SN}_1} \cup \widehat{\mathcal{SN}_2}, P, \widehat{\mathcal{DN}_1} \cup \widehat{\mathcal{DN}_2} \cup \{C_1, C_2\}) & \\ & \text{if} \quad fc_1{=}\widehat{fc_1} \text{ and } fc_2{=}\widehat{fc_2} \\ (\mathcal{SN} \cup \{P\}, n, \mathcal{DN}) & \text{if} \quad C_1 \in \mathcal{SN} \text{ or } C_2 \in \mathcal{SN} \\ & \quad\;\; \text{or } C_1, C_2 \in \mathcal{SN} \\ fc & \text{otherwise} \end{cases}$$

Note that in the first case we have a unification of the two open foci containing C_1 and C_2 as focused nodes, respectively. Thus, the number of open foci in S decreases.

(c) Let $P, C_1 \in \mathcal{N}_{\mathcal{P}}$ (sideways right) and let $\widehat{fc_1} = (\widehat{\mathcal{SN}_1}, C_1, \widehat{\mathcal{DN}_1})$:

$$fc' = \begin{cases} fc \text{ and closed} & \text{if } fc = \widehat{fc_1} \\ (\mathcal{SN} \cup \{C_2\}, n, \mathcal{DN}) & \text{if } C_1, P \in \mathcal{SN} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}} \cup \{C_1, C_2\}) & \text{if } P \in \mathcal{DN} \text{ or } n = P \\ fc & \text{otherwise} \end{cases}$$

(d) Let $P, C_2 \in \mathcal{N}_{\mathcal{P}}$ (sideways left): Symmetrical to case 5c.

(e) Let $P, C_1, C_2 \in \mathcal{N}_{\mathcal{P}}$ (closing) and let $\widehat{fc_1} = (\widehat{\mathcal{SN}_1}, C_1, \widehat{\mathcal{DN}_1})$, $\widehat{fc_2} = (\widehat{\mathcal{SN}_2}, C_2, \widehat{\mathcal{DN}_2})$:

$$fc' = \begin{cases} fc \text{ and closed} & \text{if} \quad fc = \widehat{fc_1} \text{ or } fc = \widehat{fc_2} \\ (\mathcal{SN}, n, \mathcal{DN} \cup \widehat{\mathcal{DN}_1} \cup \widehat{\mathcal{DN}_2} \cup \{C_1, C_2\}) & \\ & \text{if} \quad P \in \mathcal{DN} \text{ or } n = P_2 \\ fc & \text{otherwise} \end{cases}$$

Proof context transitions involving inference rules with different argument patterns are generalizations or combinations of the above cases.

In the case proof steps are backtracked — by removing rule applications and, when necessary, proof nodes — the transitions of proof contexts are diametric to the above transition rules and we omit the details here. □

3.3.6.2 Example

Figure 3.5 displays some steps for the proof of the example problem given in section 3.3.2. For the sake of the example the proof is not conducted in the shortest possible way but instead with some detours. In the topmost box we have the initial proof problem on the left and the associated proof context pc_1 on the right. The chronological node order is set to be $Thm <_n A_1 <_n A_2$ and the only focus context is fc. Applying the inference rule $\wedge_I$ to line Thm gives the partial proof given in the next box. The change of proof context corresponds to case 3b) in definition 3.10 and thus the new context pc_2 consists of two focus contexts fc_1 and fc_2 both containing the original assumptions as support nodes and the original theorem as derived nodes. Since we have $L_1 <_n L_2$ we automatically get $fc_1 \prec fc_2$ as order on the foci and therefore fc_2 as new active focus.

Naturally, we could have applied $\wedge_I$ in a sideways direction to immediately close the subgoal R. However, for the sake of the example we avoided this. Likewise we now introduce a step into the proof plan that is not necessary as such but serves to further illustrate the change of proof contexts. We apply the indirect rule to the focused goal L_2, thus we try to show R by deriving falsehood from $\neg R$. The transition of pc_2 to pc_3 is then according to case 2b): fc_2 gets the new hypothesis as additional support node and the old focused goal becomes an additional derived line.

In the next change $\neg_E$ is applied to the lines A_2 and L_4 to close L_3 which according to transition case 3e) closes fc_2 but leaves it unchanged. With fc_2 closed fc_1 remains as the only open focus context and thus becomes the active focus. The last change of the proof displayed in figure 3.5 is the application $\forall_E^*$ to line A_1 (again without immediately closing the subgoal L_1) which leads to the changes given in case 1a) of definition 3.10: The support nodes of both fc_1 and fc_2 are extended with L_5. We could finally close L_1 with a weaken step corresponding to a proof context transition of case 1c). However, this last step is not displayed in figure 3.5.

$A_1.$	$A_1 \vdash \forall x. \forall y. Q(x,y)$	(Hyp)		$pc_1 = (\{fc\}, <_n, \prec)$ $Thm <_n A_1 <_n A_2$ $fc = (\{A_1, A_2\}, Thm, \{\})$ fc open
$A_2.$	$A_2 \vdash R$	(Hyp)		
	$\vdots$			
$Thm.$	$A_1, A_2 \vdash Q(a,b) \land R$	$(Open)$		

$A_1.$	$A_1 \vdash \forall x. \forall y. Q(x,y)$	(Hyp)		$pc_2 = (\{fc_1, fc_2\}, <_n, \prec)$ $\ldots <_n A_2 <_n L_1 <_n L_2$ $fc_1 = (\{A_1, A_2\}, L_1, \{Thm\})$ $fc_2 = (\{A_1, A_2\}, L_2, \{Thm\})$ $fc_1 \prec fc_2$ fc_1, fc_2 open
$A_2.$	$A_2 \vdash R$	(Hyp)		
	$\vdots$			
$L_2.$	$A_1, A_2 \vdash R$	$(Open)$		
	$\vdots$			
$L_1.$	$A_1, A_2 \vdash Q(a,b)$	$(Open)$		
$Thm.$	$A_1, A_2 \vdash Q(a,b) \land R$	$(\land_I L_1 L_2)$		

$A_1.$	$A_1 \vdash \forall x. \forall y. Q(x,y)$	(Hyp)		$pc_3 = (\{fc_1, fc_2\}, <_n, \prec)$ $\ldots <_n L_2 <_n L_3 <_n L_4$ $fc_1 = (\{A_1, A_2\}, L_1, \{Thm\})$ $fc_2 = (\{A_1, A_2, L_4\}, L_3,$ $\{Thm, L_2\})$ $fc_1 \prec fc_2$ fc_1, fc_2 open
$A_2.$	$A_2 \vdash R$	(Hyp)		
$L_4.$	$A_2 \vdash \neg R$	(Hyp)		
	$\vdots$			
$L_3.$	$A_1, A_2, L_4 \vdash \bot$	$(Open)$		
$L_2.$	$A_1, A_2 \vdash R$	$(Indirect\ L_3)$		
	$\vdots$			
$L_1.$	$A_1, A_2 \vdash Q(a,b)$	$(Open)$		
$Thm.$	$A_1, A_2 \vdash Q(a,b) \land R$	$(\land_I L_1 L_2)$		

$A_1.$	$A_1 \vdash \forall x. \forall y. Q(x,y)$	(Hyp)		$pc_4 = (\{fc_1, fc_2\}, <_n, \prec)$ $\ldots <_n L_2 <_n L_3 <_n L_4$ $fc_1 = (\{A_1, A_2\}, L_1, \{Thm\})$ $fc_2 = (\{A_1, A_2, L_4\}, L_3,$ $\{Thm, L_2\})$ $fc_1 \prec fc_2$ fc_1 open fc_2 closed
$A_2.$	$A_2 \vdash R$	(Hyp)		
$L_4.$	$A_2 \vdash \neg R$	(Hyp)		
$L_3.$	$A_1, A_2, L_4 \vdash \bot$	$(\neg_E A_2 L_4)$		
$L_2.$	$A_1, A_2 \vdash R$	$(Indirect\ L_3)$		
	$\vdots$			
$L_1.$	$A_1, A_2 \vdash Q(a,b)$	$(Open)$		
$Thm.$	$A_1, A_2 \vdash Q(a,b) \land R$	$(\land_I L_1 L_2)$		

$A_1.$	$A_1 \vdash \forall x. \forall y. Q(x,y)$	(Hyp)		$pc_5 = (\{fc_1, fc_2\}, <_n, \prec)$ $\ldots <_n L_2 <_n L_3 <_n L_4 <_n L_5$ $fc_1 = (\{A_1, A_2, L_5\}, L_1,$ $\{Thm\})$ $fc_2 = (\{A_1, A_2, L_4, L_5\}, L_3,$ $\{Thm, L_2\})$ $fc_1 \prec fc_2$ fc_1 open fc_2 closed
$L_5.$	$A_1 \vdash Q(a,b)$	$(\forall^*_E A_1)$		
$A_2.$	$A_2 \vdash R$	(Hyp)		
$L_4.$	$A_2 \vdash \neg R$	(Hyp)		
$L_3.$	$A_1, A_2, L_4 \vdash \bot$	$(\neg_E A_2 L_4)$		
$L_2.$	$A_1, A_2 \vdash R$	$(Indirect\ L_3)$		
	$\vdots$			
$L_1.$	$A_1, A_2 \vdash Q(a,b)$	$(Open)$		
$Thm.$	$A_1, A_2 \vdash Q(a,b) \land R$	$(\land_I L_1 L_2)$		

Figure 3.5: Changes in proof contexts.

3.3.6.3 Guiding Ω-Ants

The purpose of foci is to separate subproblems that can be tackled independently from other parts of the proof. In this way they are used to both guide the search of argument agents as well as the user during the proof construction. In practice, argument agents restrict their search to the respective active focus; that is, support agents search among the support nodes of the active focus and consider nodes with higher chronological node order first. Goal agents have essentially just the focused node to consider. In case an inference rule has multiple conclusions, the associated goal agents also search among all focused nodes of all open focus contexts. This, however, has the effect that the support nodes in a PAI are from the active focus whereas the open nodes are from arbitrary foci and are not necessarily supported by the support nodes in the PAI. This can lead to unsound PAIs that have to be discarded. Therefore, only those PAIs are eligible that contain support nodes that are in the intersection of the support nodes of each of the open nodes.

The chronological node order is also used as an additional sorting criteria for the command agents. Suggestions with the same ratio of instantiated arguments are additionally sorted with respect to a multi-set extension of $<_n$ guaranteeing that entries with containing nodes with higher node order are preferred.

This guidance of the default mechanism serves for certain goal directed and, as we believe, more cognitively adequate guidance of the user: Firstly, keeping the suggestions restricted to the active focus until it is closed and then changing to the next open focus with respect to the order of the focus priority allows the user to focus on the solution of one subproblem before considering the next. Moreover, the order of the subproblems is chronological in the sense that the latest open subgoal is always considered first. Secondly, using a chronological order on nodes for sorting command suggestions means that facts are more likely derived when they are necessary to contribute to the solution; that is, newer nodes are more likely to be used for the task at hand than nodes derived earlier. Both restrictions are, of course, not always desirable and therefore the user can switch the active focus interactively during proof construction and select for a chosen command from all computed PAIs.

3.4 Adaptation

In this section we give an account of all possible automatic and interactive runtime adaptations of Ω-Ants. In particular, we describe how additional knowledge about the proof can be collected and used in the Ω-Ants architecture, how the suggestion mechanism can optimize itself at runtime with respect to a resource concept, and finally how the user can influence the behavior of the mechanism. These adaptations help to both narrow the search space for single agents as well as to speed up the performance of the overall mechanism.

3.4.1 Knowledge-based Adaptation

In particular proof situations it is obvious that some inference rules will never be applicable. For instance, if some subproblem is propositional it is obvious that inference rules encoding higher order rules are not necessary. Therefore, we should restrict the possible set of commands in such a proof situation by suppressing those that can definitely not be applied. This is achieved by barring the corresponding argument agents from running.

3.4.1.1 Classifying Agents

To decide whether certain commands are not worth considering for a particular subproblem at hand some knowledge has to be acquired on the subproblem and fed into the Ω-ANTS mechanism. This is done by so called *classifying agents*, which are somewhat outside the hierarchical architecture given in figure 3.3. The task of a classifying agent is to gather knowledge on a particular subgoal or subproblem, or generally about the current context of the proof.

In practice, classifying agents are realized as predicates that can be applied to single nodes or set of nodes. The agent applies the predicate to the focused subproblem whenever it has changed, that is, whenever Ω-ANTS is reset. Once the predicate succeeds the classifying agent makes its information available to other agents as elaborated in section 3.4.1.2.

Consider for instance a classifying agent whose task is to judge whether a subproblem is only propositional. It consists of a predicate that, when applied to a set of nodes comprising a subproblem, would yield true if all formulas are propositional. The agent would communicate a message indicating that the subproblem currently under consideration is just propositional to the other agents in Ω-ANTS.

3.4.1.2 Knowledge Propagation

This process of communicating knowledge to other agents is not done directly from a classifying agent to other agents but instead it is achieved via the blackboard architecture in a top-down manner. This has the advantage that a classifying agent does not need to know which other agents are currently active in the system, as this may vary dynamically.

The propagation of information is achieved as follows: A classifying agent can write information on the suggestion blackboard, there it is picked up by the command agents and written on the respective command blackboards where it is in turn read by the argument agents. Once the classifying agent cannot confirm its information, that is, when the proof state has changed such that the classifying agent's predicate does not hold anymore, it retracts its information from the suggestion blackboard. This consequently propagates throughout the mechanism similar to the adding of information.

Command and argument agents can be implemented to have information about themselves. This inherent knowledge is then compared with any information given on the blackboard the agent works for. In case the information on the blackboard indicates that the agent is useless in a particular proof situation it stops working, otherwise it further pursues its task. If an agent has suspended its task it still checks the information on the blackboard and possibly resumes its computation once the information has changed. Suspension of agents can happen on both layers in Ω-ANTS. Command agents can suspend the whole associated society of argument agents or single argument agents of a society can retire if the given information indicates that they are no longer useful.

Let us assume, for example, that our classification agent for propositional logic has just successfully diagnosed the focused subproblem as being of propositional logic. It then writes the appropriate message on the suggestion blackboard. When this message is propagated the command agent of the $ForallE*$ command will suspend its own and all its argument agents' computation since $\forall_E^*$ — as inference rule dealing with quantifier elimination — will never be applicable in the current proof situation. However, the agents working for $AndI$ will still pursue their tasks since $\wedge_I$ is a propositional logic inference rule.

3.4.2 Resource Adaptation

Even with the automatic adaptation of Ω-Ants, the number of running agents can become quite large. Moreover, some of these agents might be computationally expensive. Although the agents run concurrently, the overall computation of all agents is slowed down and thus maybe only a few useful suggestions can be computed until the user executes another command. Moreover, some agents might be so expensive that on average they do not finish their computation before a user normally executes a command. They therefore never contribute to the suggestions in the first place.

These considerations lead to a notion of *resource adaptation* in Ω-Ants that we shall present in the following. Here we adopt the notion of resource adaptation according to Zilberstein [220] meaning that agents have an explicit notion of resources themselves, enabling them to actively participate in the dynamic allocation of resources. This is opposed to the weaker notion of *resource-adapted* systems where agents only behave with respect to some initially set resource distribution. For a detailed account of the implementation of the resource mechanism in Ω-Ants see also [118].

3.4.2.1 Resource Concept

The main motivation for the resource concept of Ω-Ants is to eliminate agents that either perform fruitless computations or that use too many system resources without obvious success. In particular we want to model the following system behavior:

1. Turn off agents that in the majority of cases do not finish their computations before the user applies a command.
2. Disable agents that use too many resources, without making good suggestions in a given proof state.
3. If the user has not issued a command for a long time and most agents have already finished their computation, the system should reactivate some agents that have been turned off.
4. Agents that were turned off some time ago should be given a new chance to compute suggestions every once in a while.
5. There should always be at least a minimal number of agents running.

The idea of 1 is that, if we have an agent (e.g., a bulky external system) that always commences its computations but the user always executes a command before the agent has a chance to finish these computations, it does not have a chance to contribute to the suggestions in the first place and resources are rather wasted on this agent. Thus, it is better to stop this agent completely and free its resources for other, more effective agents.

The intention of 2 is to evaluate agents effectivity with respect to their complexity and their effectivity in a given proof state. If an agent's computation uses a lot of resources but the agent never contributes to the suggestions although it generally finishes its computation before user interaction it might be better to turn it off and free the resources for other agents. However, agents of this type should be turned off more carefully since an agent might be crucial for the success of a proof in a certain situation even though it has never computed a suggestion so far in the proof.

While 1 and 2 help to turn off possibly useless agents, 3 and 4 are a certain fairness condition for agents that have been turned off. The former is used to activate agents on a short term basis to use available system resources whenever the agents of the suggestion mechanism have completed their computations and the user still has not issued a command. The idea of the latter is to reactivate agents after some period of time since agents that might be more effective and less complex when the proof context has changed.

Lastly, 5 ensures that a certain number of agents is always running and computing suggestions. It can also be used to ensure that agents working for inference rules that are crucial to ensure completeness requirements should never retire.

We shall now define a resource concept for Ω-ANTS that reflects three major criteria:

(i) The relative complexity of an agent in a given proof situation.

(ii) The success of an agent with respect to new suggestions made.

(iii) The success of an agent with respect to the user's speed of interaction.

Criterion (i) is used to influence the *complexity rating* of an agent. The complexity rating is a value that is initially assigned to an agent and dynamically adjusted during the course of one or several proofs. Initially, all agents have the same complexity rating. However in some situation, for instance in certain mathematical domains, the complexity ratings can also be already predetermined to varying values. An argument agent influences its own complexity rating by measuring the actual CPU time it spends for each successful computation, no matter whether the computation leads to a new suggestion or not. The complexity rating can then be computed anew as average over the runs of the agent multiplied with a given weight function.

In addition to the complexity rating each agent has a *success rating*, a value that reflects the agents success in its overall computations. Initially an agents success rating is zero and it dynamically changes it by giving itself bonuses and penalties. A bonus is awarded whenever the agent successfully completes its computations by returning one or several new extended PAIs. A penalty is given if the agent's computation is not successful. Thereby we distinguish two different types of unsuccessful computation: If the agent ends its computation, but does not produce a new blackboard entry it is given a minor penalty. However, if the agent fails to finish its computation before the next proof step has been applied and Ω-ANTS's blackboards have been reset, the agent receives a major penalty. The bonus and penalty values depend on given heuristics that can vary. The success rating reflects both criteria (ii) and (iii) of our resource concept.

From the complexity and success rating each argument agent can compute its own *performance rating* by relating the two values with respect to a given heuristic function. The command agent of a society of argument agents can in turn compute the overall performance of the society by taking the average of the performance ratings of the single argument agents.

The performance rating is a measure to rate agents with respect to the *activation/deactivation threshold* of the Ω-ANTS mechanism (we shall call it activation threshold for short). The threshold is a value that determines whether an agent runs or not. In case the performance rating of an agent is below the threshold the agent can perform its computations, in case it is above the agent retires by stopping its computations. The value of the activation threshold is preset. It can be interactively changed by the user, however, and is also dynamically adapted by the running system.

3.4.2.2 Resource Adjustment

The dynamic adjustment of resources in Ω-Ants is a process of gathering resource information, reasoning on this information, and readjusting the threshold and redistributing the resources accordingly. This propagation of resource information in Ω-Ants is similar to the propagation of knowledge we described in section 3.4.1.2.

Resource information is gathered in a bottom-up direction: The single argument agents write their performance measures onto their respective command blackboards. Here the command agents can pick up the information, compute the overall performance rating of its argument agent society and propagate the information to the suggestion blackboard. Thus, the suggestion blackboard maintains information about the performance rating of all command agents and of all argument agents.

The collected information can now be used by the *resource agent*. This is an additional agent in Ω-Ants, whose task is to reason about the collected resource information and to perform readjustments if necessary. These readjustments are done in the form of penalties or bonuses for single agents, which are written to the suggestion blackboard and from there propagated throughout the system via the command agents and command blackboards. The resource agent has two ways of assigning penalties and bonuses: A penalty or bonus is either given to a single argument agent or one is assigned to a command agent, which in turn distributes it evenly amongst its argument agents. These additional penalties are a means to increase the performance of the overall system by disposing of ineffective agents more quickly to free runtime for more effective agents.

Agents that have gone below the activation threshold should have a chance of running again after a certain time. To ensure this the resource agent assigns small bonuses to them such that they again go above the threshold and can recommence their computations. Likewise the resource agent can assign larger boni to get more of the retired agents running again in a situation where for instance the proof context has changed considerably.

The resource agent can also take care that certain commands are always considered during the suggestion process. For instance, in order to ensure the completeness of a calculus some commands should never be excluded. It then has to ensure that no agent associated with one of these commands ever retires by keeping their performance rating above the activation threshold. This is done by assigning additional boni if necessary.

Another task of the resource agent is to ensure that always a certain number of agents is actually running in the system. This is achieved by regulating the activation threshold; that is, the threshold is gradually lowered, such that the minimum number of agents is running again.

All these measures of changing the resource distribution and the activation threshold of the system are more or less long term adjustments; that is, they take effect after one or several resets of the system, only. However there is also a means for short term adjustment of resources: If all active agents are finished with their computation and there is still no command being executed (i.e., the user is not interacting) the resource agent temporarily lowers the activation threshold enabling retired agents to run. However, after the next reset of the system the resource values are set to their regular values. This way the computational resources can be optimally exploited if the user's interaction interval is exceptionally long.

Finally, one important task of the resource agent is to keep track of agents that have not completed their computation after several resets of the system. They can be detected because they have not submitted any resource information over this period of time. These agents are likely candidates for either containing an erroneous predicate or for performing an expensive and maybe undecidable computation. The resource agent can reset and reinitialize such an agent, possibly giving it a penalty, in order to free those resources. This property is particularly important when external systems such as automated theorem provers are integrated into agents.

All decisions of the resource agent are subject to heuristics, which can be changed if necessary. Moreover, Ω-Ants's resource mechanism can be fine-tuned by fiddling with the variety of heuristics. We shall enumerate all possible points of interference in the next section.

Also the resource mechanism was mainly constructed to increase the systems performance in interactive reasoning, it can also be used in automation mode. Here the effects of the resource mechanism that depending on the different settings of the activation threshold as well as the resource agent, the resulting proofs can vary. We shall come back to this point in our case study in chapter 5.

3.4.3 Interactive Adaptation

This section is an overview on the different user interaction facilities, which can be used, to adapt Ω-Ants behavior at runtime.

Command selection Ω-Ants allows us to select the set of commands that is included in the suggestion process. The excluded commands are never suggested and their command agent and associated argument agents suspend their work. The user can include or exclude commands at any time and changes take effect when the mechanism is reset.

Adding new commands We can also add new commands that could not be considered by the suggestion mechanism so far, simply by specifying and loading an appropriate set of argument agents. The corresponding command agent is then automatically compiled. And, once added to the list of considered commands, suggestions for the new command are computed after the next reset. Adding completely new commands can be particularly helpful when integrating computations of a new external systems or variations of calling an already integrated external system at runtime.

Modification of societies of argument agent For already working societies of argument agents the user can specify and load new argument agents at runtime, modify the definition of already existing agents, or simply delete agents from the society. Changes to argument agent societies take again effect when Ω-Ants is reset.

Sorting heuristics The suggestion agent as well as the command agents employ heuristics to sort the entries on the respective blackboards they survey. The user can change these heuristics at runtime. For the command agents the user can either change the sorting heuristics for the whole set of command agents or for single command agents separately. Changes take effect as soon as an agent has to re-sort the entries on the blackboard it monitors, when a new entry is added.

Modification of classifying agents Similar to the modification of the argument agents the user can also add, change or remove classifying agents in order to influence Ω-Ants behavior.

Adjustment of resource adaptation The user can directly modify the resource bounds, the activation threshold and the values for penalties and boni, in order to adapt the system to particular needs. For instance, the penalty and bonus value for computations that terminate in time regardless of whether they produce a new PAI or not can be set to zero. Then a penalty is only given to agents that do not complete their computation before a reset.

Additionally, all heuristics for resource computations can be changed. Particularly interesting is here the heuristic for the resource agent. However, also the heuristics to compute the performance rating of argument agents and command agents, the weight function to compute an agent's complexity rating as well as the function to distribute boni and penalties for a command agent to its argument agent society can be fine-tuned.

3.5 Automation

Although Ω-ANTS is originally conceived as a mechanism to support a user in interactive theorem proving, there are several ways to partially or fully automate the proving process. Automation can be achieved basically in two ways:

1. Automated reasoning is performed during the suggestion process.
2. The command application itself is automated.

(1) is achieved by having commands that use automated reasoning procedures — either in the form of an internal procedure or as a call to an external automated theorem prover — available and compute their applicability. Then the automated procedure is basically called in some argument agent. For (2) we enrich the Ω-ANTS architecture with a backtracking wrapper and allow the suggestion agent to automatically execute computed command suggestion and backtrack whenever no more commands are suggested.

While (1) retains the interactive nature of the mechanism since the user can still decide which command, and therefore also which automatically derived proof should be applied, (2) actually corresponds to an automated deduction. However there is still an interaction possible since we allow for the user to execute commands even during the automated proof search. Finally, there is also a way to tightly integrate automated and interactive proof search by concentrating the user on the proof search in the actual focus, while delegating the proof of one or several background foci to the automation procedure.

3.5.1 Integrating Automated Reasoning

We first shall consider how automated reasoning procedures can be integrated into Ω-ANTS. Although we shall concentrate on the integration of external systems to achieve automation in this section everything presented here can also be extended to incorporate automated reasoning procedures internal to the ΩMEGA system into the computation. This is of course possible since we can encapsulate such an internal procedure into its own thread and if necessary terminate it via the resource mechanism.

Figure 3.6 displays two inference rules that incorporate external reasoners into ΩMEGA together with their corresponding commands on the righthand side. We have seen the *Otter* rule already earlier in this chapter. Its purpose is the application of the first order automated theorem prover OTTER to justify the conclusion C from the premises $P_1, \ldots, P_n$. The associated command has thus two arguments, the

$$\frac{P_1 \quad \ldots \quad P_n}{C}\ \mathit{Otter} \longrightarrow \frac{\mathit{Premlist}}{\mathit{Conc}}\ \mathit{Otter}$$

$$\frac{A \quad B \Rightarrow C}{C}\ \mathit{MP\text{-}mod\text{-}ATP}(A \Rightarrow B) \longrightarrow \frac{\mathit{Left} \quad \mathit{Impl}}{\mathit{Conc}}\ \mathit{MpModAtp}$$

Figure 3.6: Inference rules to apply external reasoners and their commands.

$\mathfrak{S}^{\{Premlist\}}_{\{\},\{Conc\}}$	=	$\{$*Premlist*: All support nodes containing first order formulas$\}$
$\mathfrak{S}^{\{Conc\}}_{\{Premlist\},\{\}}$	=	$\{$*Conc*: *Conc* can be justified with OTTER from *Premlist*$\}$

Table 3.3: Argument agents of the *Otter* command.

conclusion and a list of premises. The second inference rule *MpModAtp* describes a situation where an external reasoner — in this case an automated theorem prover — is used within an inference modulo, in our particular case modus ponens modulo the validity of an implication (to be checked by an automated theorem prover). The modulo implication is given as a parameter of the inference rule. The corresponding command, however, has only three arguments for the proof nodes involved, since the parameter can be computed from the given arguments *Left* and *Impl*. Possible instantiations for the arguments are for instance the following:

$$\begin{array}{rcl} \mathit{Left} & \leftarrow & \forall x_{\bullet} P(x) \wedge Q(x) \\ \mathit{Impl} & \leftarrow & P(a) \Rightarrow R(a) \\ \mathit{Conc} & \leftarrow & R(a) \end{array}$$

Then the modulo implication the theorem prover has to test for validity in order for the *MP-mod-ATP* rule to be applicable is $(\forall x_{\bullet} P(x) \wedge Q(x)) \Rightarrow P(a)$. This kind of modulo reasoning can also be done with different external reasoners, for instance, with respect to the simplification of a computer algebra system.

Right now, we are not concerned with correctness issues for the integration of the external systems. However, since we are working in the ΩMEGA environment we can make use of the work already done in this area that ensures the correctness by translating proofs or computations from external reasoners into primitive inference steps of ΩMEGA [146].

The agents determining the applicability of the *Otter* and *MpModAtp* commands are given in the figures 3.3 and 3.4, respectively. The agents for the *Otter* command are straightforward. We have one support agent that searches among all the support nodes of the active focus those that contain first order formulas; that is, the agent actually filters out those lines containing higher order formulas. The second agent $\mathfrak{S}^{\{Conc\}}_{\{Premlist\},\{\}}$ actually encapsulates the application of the OTTER theorem prover. Its task is to check whether the open node of the active focus can be derived from the given premises.

The *MpModAtp* command has seven agents where some of them cater for the possibility that some of the arguments are supplied by the user. The computations of the automated theorem prover is embedded into the last three of these agents. The agents $\mathfrak{S}^{\{Impl\}}_{\{Left\},\{Conc\}}$ and $\mathfrak{S}^{\{Impl\}}_{\{Left,Conc\},\{\}}$ search for an appropriate implica-

$\mathfrak{S}^{\{Conc\}}_{\{\},\{Impl\}}$	$=\{$*Conc*: The focused node of the active focus$\}$
$\mathfrak{S}^{\{Conc\}}_{\{Impl\},\{\}}$	$=\{$*Conc*: *Conc* is equal to the succedent of *Impl*$\}$
$\mathfrak{S}^{\{Impl\}}_{\{\},\{Left,Conc\}}$	$=\{$*Impl*: *Impl* is implication$\}$
$\mathfrak{S}^{\{Impl\}}_{\{Conc\},\{Left\}}$	$=\{$*Impl*: *Impl* is implication with succedent equal to *Conc*$\}$
$\mathfrak{S}^{\{Impl\}}_{\{Left\},\{Conc\}}$	$=\{$*Impl*: *Impl* is implication and its antecedent can be derived from $Left$ using an ATP$\}$
$\mathfrak{S}^{\{Impl\}}_{\{Left,Conc\},\{\}}$	$=\{$*Impl*: *Impl* is an implication, the succedent is equal to *Conc* and the antecedent can be derived from $Left$ using an ATP$\}$
$\mathfrak{S}^{\{Left\}}_{\{Impl\},\{\}}$	$=\{$$Left$: Antecedent of *Impl* is derivable from $Left$ with an ATP$\}$

Table 3.4: Argument agents of the *MpModAtp* command.

tion by checking with the theorem prover that the antecedent follows from the already given instantiation of $Left$. Agent $\mathfrak{S}^{\{Left\}}_{\{Impl\},\{\}}$ on the other hand searches for the instantiation of $Left$ given the implication by checking the derivability of the antecedent.

3.5.2 Automating the Command Application

The Ω-Ants suggestion mechanism can be automated into a full-fledged proof search procedure by embedding the execution of suggested commands into a backtracking wrapper. The algorithm itself is given in table 3.5 and its flowchart is given in figure 3.7.

The basic automation performing a depth first search is straightforward: The suggestion mechanism waits until all agents have performed all possible computations and no further suggestions will be produced and then executes the heuristically preferred suggestion (1a&2). When a proof step is executed and the proof is not yet finished, the remaining suggestions on each suggestion blackboard are pushed on the backtracking stack (3&4). In case no best suggestion could be computed Ω-Ants backtracks by popping the first element of the backtrack stack and re-instantiating its values on the blackboards (7a&7d).

The restriction to a depth first search strategy is forced by the nature of Ωmega's proof plan data structure $\mathcal{PDS}$, which does not yet allow for storing parallel proof attempts. Therefore, search strategies like breadth first or best first are not yet supported.

The simple automation loop is complicated by the distinct features of Ω-Ants:

(i) Certain agents can perform infinite or very costly computations.

(ii) Commands can be executed by the user in parallel to the automation.

(iii) The components of Ω-Ants can be changed at runtime.

Furthermore, the automation can also be suspended and revoked especially in order to perform the latter two interaction possibilities in a coordinated way.

1. Wait for suggestions to be made until:
   (a) no further suggestions can be found,
   (b) a time bound is exceeded,
   (c) or the user has executed a command. Then goto step 3.
2. If there are suggestions on the command blackboard execute the heuristically preferred suggestion, otherwise goto step 7.
3. If a proof has been found, quit with $\boxed{\text{success}}$.
4. Push the history on the blackboards onto the backtrack stack.
5. Execute changes of agents and heuristics if there were any.
6. Re-initialize all blackboards and goto step 1.
7. If there have been changes of agents or heuristics goto step 5, otherwise do a backtracking step by popping the backtrack stack:
   (a) if stack is empty, stop with $\boxed{\text{failure}}$,
   (b) if the popped step has been executed due to expired time but has not been re-instantiated before, goto step 1,
   (c) if the step has been introduced after command execution by the user, goto step 1,
   (d) in all other cases goto step 2.

Table 3.5: Algorithm for the automation wrapper of Ω-Ants.

We avoid that Ω-Ants is paralyzed by agents that get stuck in infinite computations by giving a time limit after which the best command, suggested so far, is executed (see step 1b). However, such a proof step is treated special when backtracking, since then the blackboards will be re-instantiated with all the values of the proof step, containing the executed command as well. This way there is a second chance for agents that could not contribute the first time to add information. But should the step be executed in the same form once more, it will then be backtracked in a regular manner (7b). The question how the Ω-Ants theorem prover can avoid to get lost on an infinite branch in the search space without ever backtracking will be addressed in the completeness discussion in section 3.6.

If a command has been executed by the user the loop proceeds immediately with saving the blackboards' history without executing another command (1c). When backtracking the whole history on the last step is re-instantiated onto the blackboards, possibly containing also the command executed by the user, in order not to loose possible proofs (1c&7c).

One main feature of Ω-Ants is its runtime adaptability by adding or deleting agents or changing the heuristics for sorting command and suggestion blackboards. These changes also take effect when running the automation wrapper, precisely in the steps 5&7. The automation wrapper can be suspended by the user at any time, for instance, in order to analyze the current proof state and to add, change or remove certain agents from the suggestion mechanism. It can then be resumed using all the information computed so far.

For the automation to make sense it is necessary to carefully choose the set of

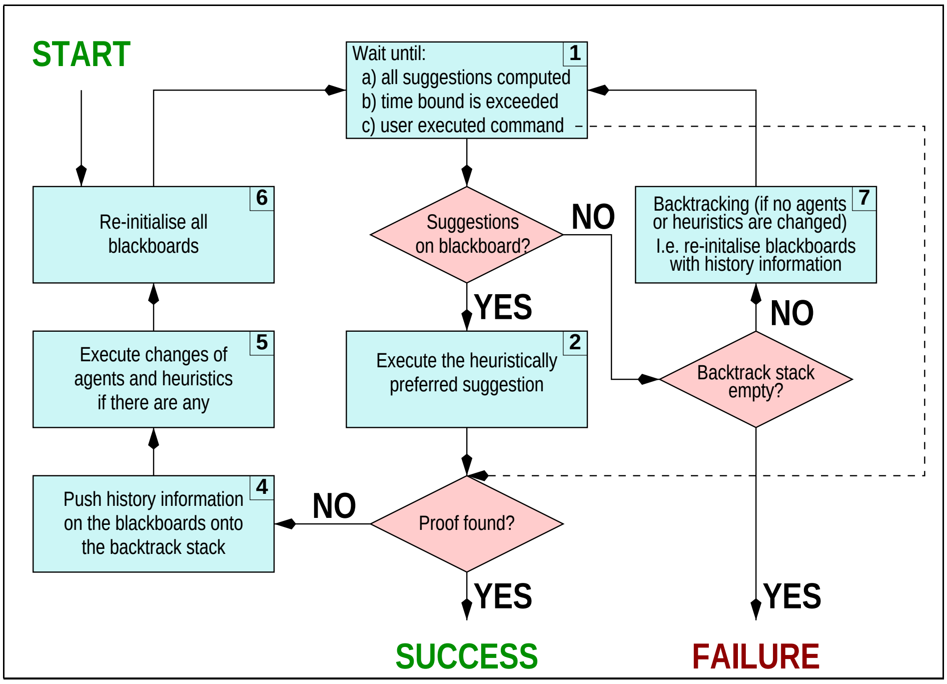

START
Wait until:
a) all suggestions computed
b) time bound is exceeded
c) user executed command
1
Re-initialise all blackboards
6
Suggestions on blackboard?
NO
Backtracking (if no agents or heuristics are changed) I.e. re-initalise blackboards with history information
7
YES
NO
Execute changes of agents and heuristics if there are any
5
Execute the heuristically preferred suggestion
2
Backtrack stack empty?
Push history information on the blackboards onto the backtrack stack
4
NO
Proof found?
YES
YES
SUCCESS
FAILURE

Figure 3.7: The main loop of the automation wrapper.

inference rules that are considered in order to achieve a benevolent search behavior. In chapter 5 we shall see a case study, in which the normal form natural deduction calculus Nic [55] is modeled in Ω-Ants and the automation wrapper is applied to perform goal directed search.

3.5.3 Automation as a Background Process

The decomposition of the whole proof into explicitly maintained subproblems by the focus contexts introduced in section 3.3.6 enables the system to perform certain automated tasks in the background while the user works on a different sub-problem interactively. More concretely, the user concentrates on the currently focused sub-problem (i.e., the active focus) where the interaction is supported by the suggestions of Ω-Ants. In the meantime the subproblems of one or more different open foci can be tackled automatically in the background.

A straightforward way of achieving this is to delegate a subproblem to one or several external reasoning systems. Once an external system has found a proof for the subproblem, this subproof is inserted either immediately or after querying the user first. This way of distributing the reasoning process is essentially independent from the Ω-Ants architecture.

However, apart from the straightforward distribution, we can also use Ω-Ants to directly automate search in the background. This scenario works in the following way: While the user works on the subproblem in the active focus, Ω-Ants computes suggestions for the interaction. In parallel Ω-Ants also computes suggestions for other focus contexts, which can then be automatically applied. For both reasoning processes — the interactive user session and the automated background reasoning — agents can also be used that in turn use external reasoners during their search as discussed in previous section. However, to achieve this kind of distribution we need to expand Ω-Ants in a suitable way.

The naïve approach for this distribution were to simply duplicate the Ω-Ants mechanism for each focus, which is considered for background automation. But this would simply overwhelm the overall system with too many concurrent processes rendering all the improvements introduced in section 3.4 pointless. Therefore, the distribution is accomplished by duplicating only the top level of the Ω-Ants architecture.

In practice, this works as follows: Besides the active focus certain other foci are chosen for background automation. We call these foci the *background foci*. Their number as well as their selection can be adjusted interactively. For each of the background foci a different suggestion blackboard together with a suggestion agent is created.

Argument agents then always search for both the active focus and the background foci. This is essentially only an extension of the general search function for agents. In order not to compute incorrect results, PAIs on the command blackboards are supplied with the information on the particular focus it was computed for. Thus, when an predicate agent picks up a PAI that has been computed for a focus fc the agent will only search in the respective nodes of fc for completions of the PAI.

The command agents then sort the entries of their command blackboards according to the respective foci they belong to and pass the best entries for each focus — active focus and background foci — to the respective blackboards. This is achieved by supplying an appropriate search function for the command agents. The separate suggestion agents can then sort their suggestions with respect to their own heuristics and pass them, in case of the active focus, to the user or, in case of the background foci, to the automation wrapper. In the latter case the automation wrapper has to keep a separate record for each background focus. Since the automation loop of Ω-Ants runs in its own process separate bookkeeping for separate background foci is done by having a different automation wrapper for each single background process.

The knowledge and resource adaptation in the extended system proceeds as follows: The classifying agents are enabled to perform their classification task in all considered foci and write there information enriched with the focus information to the suggestion blackboard. This way command and argument agents can decide in which of the foci they can search and in which not.

On the contrary for the resource adjustment the resource agent only takes data into account, which has been collected for the active focus. This is due to the fact that one main factor in the resource concept is the interaction interval of the user, which should not be obscured with data won in parallel automated processes. However, in order to have both an efficient suggestion mechanism for the interaction part and effective automation of proofs in background foci the resource concept will have to be expanded accordingly.

3.6 Theoretical Considerations

In this section we introduce and discuss some notions that are necessary to characterize and guarantee completeness and soundness of a theorem prover based on Ω-Ants with respect to the underlying calculus. We first present how the predicate of our agents can be formalized within Church's simply typed lambda calculus. Thereby we are interested in modeling the search and communication properties of single agents and the resulting behavior of the whole agent society. We are, however, not interested in formally modeling the computational aspects within an agent society since this would require the formalization of temporal properties and therefore the use of temporal logics. Neither do we model the resource management or the self-evaluation of an agent's performance. But the presented theoretical considerations can provide feedback for designing adequate heuristics for Ω-Ants in particular with respect to the question which agents should never be turned of by the resource agent in order to guarantee the effectiveness of an agent society or the completeness of a particular calculus.

When modeling the search properties of an agent we have generally to deal with complex predicates and functions that serve as interfaces to the environment (e.g., by serving as search criteria for proof lines). We shall model only some simple predicates, but generally we abstract from the layer of actual computation. For instance, we do not want to model exactly Otter's computations in lambda calculus, although surely possible because of Church's thesis, since this is beyond the scope of this thesis. (For a more thorough introduction to computations in lambda calculus see [203]). The discussion in the remainder of this section is thus rather example driven to give an intuition for the properties that need to be considered.

Given a theoretically complete calculus, how can it be modeled in Ω-Ants such that completeness is still assured in the mechanism? Note that we do not address the theoretical completeness of the underlying calculus itself, in fact we do not even need to specify here what particular logic and calculus we are interested in. We rather aim to ensure that each calculus rule application that is theoretically possible in a given proof state can indeed be determined and suggested by the Ω-Ants mechanism. In particular we will discuss two different notions of completeness in this sense, namely *interaction completeness* and *automation completeness*. This is due to the twofold bias of the Ω-Ants system as a suggestion mechanism and as an automated theorem prover. Naming these properties also 'completeness' might be slightly misleading, however, automation (interaction) completeness of the agent societies involved taken together with the 'theoretical (logical) completeness' of a calculus implies that a complete proof search is actually supported by Ω-Ants.

Theoretical completeness investigations typically assume non-limited resources like computation time and space. In our case the resources available to the Ω-Ants-system in-between the command executions are crucial with respect to completeness as well. However, for the time being we neglect points possibly interfering with this assumption, in particular cases 1(b) or 1(c) of the prover's main loop in figure 3.7 and the existence of agents with calls to undecidable procedures such as the Otter agent in section 3.5.1.

3.6.1 Formalization

In order to formalize the predicates of our agents we first need to define the notion of conditional branching in lambda calculus. Therefore, we define an if-then-else predicate by using the description operator introduced in definition 2.4 as follows:

$$\textit{if-then-else} \equiv \lambda P_o\textbf{.}\lambda x_\alpha\textbf{.}\lambda y_\alpha\textbf{.}\iota z_\alpha\textbf{.}[P \land z = x] \lor [\neg P \land z = y] \tag{3.1}$$

According to definition (3.1) *if-then-else* is a predicate with three arguments of the form (*if-then-else* P_o x_α y_α) meaning that if P holds the predicate returns x and if $\neg P$ holds it returns y. As syntactical sugar we write generally $if\ P\ then\ x\ else\ y$. The semantics of *if-then-else* can be derived from the semantics of the description operator and of lambda abstraction given in definition 2.15 in chapter 2.1.2.

Before we can formalize the predicates for agents we have to develop a formal notion for the PAIs. Since a PAI is basically a set of mappings from formal arguments to their instantiation, we can model it as a list of pairs. And, to begin with, we define a polymorphic constant ϵ_β standing for the empty argument. Then we formally define our mapping operator ':'.

$$: \;\equiv\; \lambda x_\alpha\text{.}\lambda y_\alpha\text{.}\lambda g_{\alpha\alpha o}\text{.}g(x,y) \tag{3.2}$$

We shall write the mappings of the PAI in infix notation as $x{:}y$ instead of $:(x,y)$.

To formalize lists we need to define a construction operator '#' as being

$$\# \;\equiv\; \lambda a_\alpha\text{.}\lambda b_\beta\text{.}\lambda h_{\alpha\beta o}\text{.}h(a,b) \tag{3.3}$$

Note that the constructor is very similar to the pairing operator ':', albeit more general since we do not require the elements a and b to be of the same type. Since lists are recursive structures and therefore also need to be accessed recursively we need to define the empty list as the polymorphic constant $[]_{(\gamma\gamma o)o}$. Examples of lists are then $\#(a,[])$, $\#(b,\#(a,[]))$, etc. which we shall generally write as $(a\#[])$, $(b\#a\#[])$, ... When it is obvious from the context that we are dealing with lists we will sometimes even omit the empty list symbol. Notice, that we explicitly allow for the elements of a list to be of different types.

We can now formally define PAIs: Given a command $\mathcal{C}$ with formal arguments $A_1,\ldots,A_n$, we define constants $a^1_{\alpha_1},\ldots,a^n_{\alpha_n}$, where the types α_i correspond to the types of the required arguments for the A_i. We then denote the *abstract* PAI of $\mathcal{C}$ as the lambda expression

$$\lambda l^1{}_{\alpha_1}\text{.}\lambda l^2{}_{\alpha_2}\text{.}\ldots\lambda l^n{}_{\alpha_n}\text{.}[a_1{:}l_1]\#[a_2{:}l_2]\#\ldots\#[a_n{:}l_n]\#[].$$

A *concrete* PAI can then be constructed by applying the abstract PAI to a set of actual arguments.

We shall now define a function with which we can access the single mappings in the PAI by name of the formal argument. In order to do that we need functions both to extract the elements of the list representing the PAI and to project the first and second elements of the mappings. We first define the projection function for the mappings:

$$\begin{aligned} Proj1 &\equiv \lambda p_{(\alpha\alpha o)o}\text{.}\iota x_\alpha\text{.}\exists y_\alpha\text{.}p = x{:}y && (3.4)\\ Proj2 &\equiv \lambda p_{(\alpha\alpha o)o}\text{.}\iota y_\alpha\text{.}\exists x_\alpha\text{.}p = x{:}y && (3.5)\end{aligned}$$

When applied to a mapping of the form $x{:}y$, $Proj1$ and $Proj2$ return the elements x and y, respectively.

To recursively extract the single elements of a given list we define two functions, *First* and *Rest*. Both are similar to the two projection function of ':', albeit more general with respect to the types of the elements.

$$\begin{aligned} First &\equiv \lambda c_{(\alpha\beta o)o}\text{.}\iota a_\alpha\text{.}\exists b_\beta\text{.}c = a\#b && (3.6)\\ Rest &\equiv \lambda c_{(\alpha\beta o)o}\text{.}\iota b_\beta\text{.}\exists a_\alpha\text{.}c = a\#b && (3.7)\end{aligned}$$

Since we can now access the elements of a PAI we can define a function that can pick the single elements with respect to the name of the formal argument. Note that this is possible since we require the formal arguments of a command to have unique names.

$$\begin{aligned} Pick \equiv \lambda PAI_{(\alpha\beta o)o}\text{.}\lambda B_\gamma\text{.}&if\ [PAI \neq []]\ then && (3.8)\\ &\quad if\ Proj1(First(PAI)) = B\ then\\ &\qquad Proj2(First(PAI))\\ &\quad else\ Pick(Rest(PAI),B)\\ &else\ \epsilon\end{aligned}$$

The *Pick* function takes two arguments, a list representing the partial argument instantiation PAI and a formal argument B, and returns the instantiation of the formal argument. Its definition is recursive; that is, it calls itself on the tail of the given list if the head element does not contain the formal argument in question. We ensure termination with the outermost if statement since in case we reach the empty list representing the end of the PAI *Pick* returns the empty argument ϵ. This is due to two reasons: Firstly, it ensures that the definition of *Pick* is well typed, and secondly, we do not have to be concerned whether all not yet instantiated formal arguments are actually given in the PAI. Termination can easily be proven by an induction on the length of a given list, however, we omit this here. To abbreviate the *Pick* function we shall generally write $Pick_B(PAI)$.

We have now the formal apparatus available to actually formalize our argument agents. However, as already mentioned, we are interested in a formalization down to a certain level, only. Further formalization goes beyond the scope of this thesis and will be subject of future work. The following are the formalization of both predicate and function agents.

$$
\begin{aligned}
\mathfrak{G}^{\{G_1,G_2,\ldots\}}_{\{D_1,D_2,\ldots\},\{E_1,E_2,\ldots\}} \equiv\ & \lambda\, PAI_{(\alpha\beta o)o}\text{.}\lambda L_o\text{.}\\
& \quad if\ [[Pick_{G_1}(PAI) = \epsilon] \wedge [Pick_{G_2}(PAI) = \epsilon] \wedge \ldots\\
& \qquad\quad [Pick_{D_1}(PAI) \neq \epsilon] \wedge [Pick_{D_2}(PAI) \neq \epsilon] \wedge \ldots\\
& \qquad\quad [Pick_{E_1}(PAI) = \epsilon] \wedge [Pick_{E_2}(PAI) = \epsilon] \wedge \ldots]\ then\\
& \qquad if\ [\mathbb{P}_1(L, Pick_{D_1}(PAI), Pick_{D_2}(PAI), \ldots)]\ then\\
& \qquad\quad [G_1{:}L]\#[G_2{:}\mathbb{P}_2(L, Pick_{D_1}(PAI), \ldots)]\#\ldots\\
& \qquad\quad [D_1{:}Pick_{D_1}(PAI)]\#[D_2{:}Pick_{D_2}(PAI)]\#\ldots\\
& \qquad\quad [E_1{:}\epsilon]\#[E_2{:}\epsilon]\#\ldots\\
& \qquad\quad [R_1{:}Pick_{R_1}(PAI)]\#[R_2{:}Pick_{R_2}(PAI)]\#\ldots\#[]\\
& \qquad else\ PAI\\
& \quad else\ PAI
\end{aligned}
$$

Although the above agent is a goal agent the formalization for support agents is exactly the same. The agent's goal set is $\{G_1, G_2, \ldots\}$, its dependency set is $\{D_1, D_2, \ldots\}$ and its exclusion set is $\{E_1, E_2, \ldots\}$. The agent takes a partial argument instantiation $PAI_{(\alpha\beta o)o}$ and a formula L_o which corresponds to the formula the agent seeks. The agent's formalization itself essentially consists of two nested *if-then-else* expressions: The first is to check for the applicability of the agent to a PAI given on the blackboard and the second is to model the actual search predicate of the agent. Thus, the semaphore of the first *if-then-else* ensures that the given PAI is suitable for the condition of the argument agent by checking that all formal arguments of the goal and exclusion set are not yet instantiated in the given PAI and, conversely, that actual arguments for all elements of the dependency set are already given. The former corresponds to the first and third line and the latter to the second line of the conjunction.

The second *if-then-else* then checks whether the formula in question corresponds to one the agent actually seeks. This is indicated by the predicate $\mathbb{P}_1$, which is the actual interface to the computational part of the agents. The predicate takes the considered formula L as well as all the instantiations of the necessary arguments $D_1, D_2, \ldots$ in the given PAI as arguments. For simple cases, for instance, when dealing with rules of the ND calculus, we can explicitly give the predicate by stating the required pattern of the sought formula. But in the general case we give here only an abstract characterization of the predicate, such as 'L is higher order'. In case the conditions of both *if-then-else* predicates hold a new PAI is constructed in which the abstract argument G_1 is bound to the formula L and instantiations for the remaining goal arguments are computed. These computations are again given in terms of a predicate $\mathbb{P}_i$, whose arguments are the formula L and the actual arguments of the dependency set. The rest of the PAI is then constructed by adding the original instantiations of the formal arguments of both the dependency set and the exclusion set. Additionally, in the final line of the newly constructed PAI we carry over the rest elements of the original PAI; that is, all elements that occur in neither the goal, the dependency, nor the exclusion set of the agent are simply copied.

In the case either of the *if-then-else* tests fails — if the PAI is not of the right form or the considered formula does not fulfill the requirements — the original PAI is returned.

Note that we do not express that the agent is a goal agent within the lambda expression and that L_o should be a formula of an open node. For our current considerations this is not necessary. However, in a further development of the formalization one should consider the use of sorts or annotations in order to express these properties. Note also that we again encounter the problem of separating nodes and their formulas as discussed earlier. The expression talks about a given formula only and constructs thus a PAI that could be ambiguous with respect to which proof node is actually meant since we can have the same formula in several nodes. Despite this ambiguity, the formalization is precise enough for the further considerations in this section.

Next we examine the general form of function agents, which are of slightly simpler built than argument agents.

$$\begin{aligned}
\mathfrak{F}^{\{G_1,G_2,\ldots\}}_{\{D_1,D_2,\ldots\},\{E_1,E_2,\ldots\}} \equiv \lambda\, PAI_{(\alpha\beta o)o}\centerdot\ & \\
& if\ [[Pick_{G_1}(PAI)=\epsilon]\wedge[Pick_{G_2}(PAI)=\epsilon]\wedge\ldots \\
& \quad [Pick_{D_1}(PAI)\neq\epsilon]\wedge[Pick_{D_2}(PAI)\neq\epsilon]\wedge\ldots \\
& \quad [Pick_{E_1}(PAI)=\epsilon]\wedge[Pick_{E_2}(PAI)=\epsilon]\wedge\ldots]\ then \\
& \quad [G_1{:}\mathbb{P}_1(Pick_{D_1}(PAI),Pick_{D_2}(PAI),\ldots)]\#\ldots \\
& \quad [D_1{:}Pick_{D_1}(PAI)]\#[D_2{:}Pick_{D_2}(PAI)]\#\ldots \\
& \quad [E_1{:}\epsilon]\#[E_2{:}\epsilon]\#\ldots \\
& \quad [R_1{:}Pick_{R_1}(PAI)]\#[R_2{:}Pick_{R_2}(PAI)]\#\ldots\#[] \\
& else\ PAI
\end{aligned}$$

The lambda expression of a function agent is simpler since it always performs the desired computation if the PAI is of the right form. Therefore, only the validity of the PAI has to be checked. Since the PAI is the only lambda bound variable we can also omit the second *if-then-else* statement completely. The computation for all goal arguments, even G_1, is performed when assembling the new PAI. Analogously to the predicate agents the computations are hidden in the abstract predicates $\mathbb{P}_i$. However, in the case of function agents the predicates range over the elements of the dependency set, only.

Since the general formalization of argument agents is not necessarily intuitive we give the formalization of the agent society of the *AndI* command as an example. This example has also the advantage that we can easily formalize the occurring test predicates of our agents. As constants representing the formal arguments of the *AndI* command we use $Conj_o, LConj_o$, and $RConj_o$ and have thus an abstract PAI of the form

$$\lambda A_o\centerdot\lambda B_o\centerdot\lambda C_o\centerdot[Conj{:}A]\#[LConj{:}B]\#[RConj{:}C]\#[]$$

The single argument agents are then formally defined as:

$$
\begin{array}{lll}
\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}} & \equiv & \lambda\, PAI_{(\alpha\beta o)o}.\lambda L_o. \\
 & & \quad if\ [[Pick_{Conj}(PAI)=\epsilon] \wedge [Pick_{LConj}(PAI)=\epsilon] \\
 & & \qquad\qquad\qquad\qquad \wedge[Pick_{RConj}(PAI)=\epsilon]]\ then \\
 & & \qquad if\ [\exists A_o.\exists B_o.L=[A\wedge B]]\ then \\
 & & \qquad\quad [Conj{:}L]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[] \\
 & & \qquad else\ PAI \\
 & & \quad else\ PAI
\end{array}
$$

$\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$ takes a partial argument instantiation $PAI_{(\alpha\beta o)o}$ and a formula L_o. The semaphore of the first *if-then-else* ensures that the given PAI is suitable for the condition of the argument agent. Since the $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$ agent searches an instantiation for the *Conj* argument if neither the *LConj* nor the *RConj* argument are already instantiated, the condition checks that all formal arguments are instantiated with the empty actual argument, only. The second *if-then-else* then checks whether the formula in question is actually a conjunction with the condition $\exists A_o.\exists B_o.L=[A\wedge B]$. Compared to the general form of predicate agents given above, the predicate $\mathbb{P}_1$ corresponds to the term $\lambda X_o.\exists A_o.\exists B_o.X=[A\wedge B]$, which is applied to L as its only argument. In case the conditions of both *if-then-else* predicates hold a new PAI is constructed, in which the abstract argument *Conj* is bound to the formula L and all other abstract arguments are again bound to ϵ. In all other cases the original PAI is returned.

The second goal agents for the *AndI* command is formalized as follows:

$$
\begin{array}{lll}
\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}} & \equiv & \lambda\, PAI_{(\alpha\beta o)o}.\lambda L_o. \\
 & & \quad if\ [[Pick_{Conj}(PAI)=\epsilon] \wedge [Pick_{LConj}(PAI)\neq\epsilon] \\
 & & \qquad\qquad\qquad\qquad \wedge[Pick_{RConj}(PAI)=\epsilon]]\ then \\
 & & \qquad if\ [\exists B_o.L=[Pick_{LConj}(PAI)\wedge B]]\ then \\
 & & \qquad\quad [Conj{:}L]\#[LConj{:}Pick_{LConj}(PAI)]\#[RConj{:}\epsilon]\#[] \\
 & & \qquad else\ PAI \\
 & & \quad else\ PAI
\end{array}
$$

The $\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}}$ is an agent that depends on an already instantiated actual argument *LConj*. Therefore, one of the conditions ensuring the suitability of the PAI is $Pick_{LConj}(PAI)\neq\epsilon$. The actual instantiation for *LConj* is also used in the search predicate $\exists B_o.L=[Pick_{LConj}(PAI)\wedge B]$ and it is also carried into the new PAI that is constructed.

The rest of the goal agents for *AndI* are:

$$
\begin{array}{lll}
\mathfrak{G}^{\{Conj\}}_{\{RConj\},\{LConj\}} & \equiv & \lambda\, PAI_{(\alpha\beta o)o}.\lambda L_o. \\
 & & \quad if\ [[Pick_{Conj}(PAI)=\epsilon] \wedge [Pick_{LConj}(PAI)=\epsilon] \\
 & & \qquad\qquad\qquad\qquad \wedge[Pick_{RConj}(PAI)\neq\epsilon]]\ then \\
 & & \qquad if\ [\exists A_o.L=[A\wedge Pick_{RConj}(PAI)]]\ then \\
 & & \qquad\quad [Conj{:}L]\#[LConj{:}\epsilon]\#[RConj{:}Pick_{RConj}(PAI)]\#[] \\
 & & \qquad else\ PAI \\
 & & \quad else\ PAI
\end{array}
$$

$$
\begin{array}{l}
\mathfrak{G}^{\{Conj\}}_{\{LConj,RConj\},\{\}} \equiv \\
\qquad \lambda\, PAI_{(\alpha\beta o)o}.\lambda L_o. \\
\qquad\quad if\ [[Pick_{Conj}(PAI)=\epsilon] \wedge [Pick_{LConj}(PAI)\neq\epsilon] \\
\qquad\qquad\qquad\qquad\qquad \wedge[Pick_{RConj}(PAI)\neq\epsilon]]\ then
\end{array}
$$

```
        $if\ [L = [Pick_{LConj}(PAI) \wedge Pick_{RConj}(PAI)]]\ then$
            $[Conj{:}L]\#[LConj{:}Pick_{LConj}(PAI)]\#[RConj{:}Pick_{RConj}(PAI)]\#[]$
        $else\ PAI$
    $else\ PAI$
```

The following two formalizations are for the support agents of the *AndI* command. Their particularity is that their applicability does not depend on the constellation of all abstract arguments but only on a subset. For instance $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ does not include the *RConj* in the goal, dependency, or exclude set. This is expressed in the simpler condition of the first *if-then-else* statement, which expresses only a constraint with respect to the instantiations of the *Conj* and *LConj* abstract arguments. It is, however, ensured that whatever instantiation for *RConj* the PAI already contains is carried into the newly constructed PAI.

```
$\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}} \equiv$
    $\lambda\ PAI_{(\alpha\beta o)o}\centerdot \lambda L_o\centerdot$
        $if\ [[Pick_{Conj}(PAI) \neq \epsilon] \wedge [Pick_{LConj}(PAI) = \epsilon]]\ then$
            $if\ [\exists B_o\centerdot [L \wedge B] = Pick_{Conj}(PAI)]\ then$
                $[Conj{:}Pick_{Conj}(PAI)]\#[LConj{:}L]\#[RConj{:}Pick_{RConj}(PAI)]\#[]$
            $else\ PAI$
        $else\ PAI$
```

The search predicate of the agent is $\exists B_o\centerdot [L \wedge B] = Pick_{Conj}(PAI)$. It expresses that L is the left conjunct of a conjunction given as instantiation of $Conj$.

```
$\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}} \equiv$
    $\lambda\ PAI_{(\alpha\beta o)o}\centerdot \lambda L_o\centerdot$
        $if\ [[Pick_{Conj}(PAI) \neq \epsilon] \wedge [Pick_{RConj}(PAI) = \epsilon]]\ then$
            $if\ [\exists A_o\centerdot [A \wedge L] = Pick_{Conj}(PAI)]\ then$
                $[Conj{:}Pick_{Conj}(PAI)]\#[LConj{:}Pick_{LConj}(PAI)]\#[RConj{:}L]\#[]$
            $else\ PAI$
        $else\ PAI$
```

In the sequel we shall use the formalization we have given here to discuss and illustrate two notions of completeness, namely automation and interaction completeness.

3.6.2 Automation Completeness

The idea of *automation completness* is that given a theoretically complete calculus how can it be modeled in Ω-Ants such that completeness is still assured in the mechanism when using the automatic proof search? As a calculus we consider the first order fragment of Ωmega's calculus together with the tertium non datur axiom. This fragment is complete for first order logic as is for instance shown be Byrnes in [55]. Byrnes also gives a special purely backward search procedure for this calculus called Nic, which preserves completeness. We shall outline Nic and its modeling in Ω-Ants in chapter 5.

In this section we introduce the basic concepts necessary to define automation completeness. We shall, however, not give detailed proofs for all calculus rules and agents involved, but only use the society of agents for the *AndI* command to exemplify the notions.

Automation completeness depends in the first place on the *suggestion completeness* of the argument agent societies associated with each rule.

Definition 3.11 (Suggestion completeness): A society of suggestion agents working for a single command C is called *suggestion complete* with respect to a given calculus, if in any possible proof state all PAIs of a command necessary to ensure completeness of the calculus can be computed by the mechanism. □

Assuming, as already mentioned, that our agents always have sufficient time to perform their computations, suggestion completeness requires that each particular agent society consists of *sufficiently* many individual suggestion agents and that their particular definitions are *adequate* for computing arguments that comply with the form and combination of the respective calculus rule's arguments. *Adequacy* basically excludes wrong agent specifications, while *sufficiency* refers to the ability of an agent society to cooperatively compute each applicable PAI in a given proof state.

In order to ensure that computed PAIs are actually propagated in the mechanism we define the notion of *non-excluding* agent.

Definition 3.12 (Non-excluding): A *command agent* is *non-excluding* if it indeed always reports at least one selected entry from the associated command blackboard to the suggestion blackboard as soon as the former contains some applicable PAIs.

The *suggestion agent* is *non-excluding* if it always reports the complete set of entries on the command blackboard to the automation wrapper. □

Additionally, we have to guarantee that the proof search is organised in a *fair* way by ensuring that the execution of an applicable PAI suggested within a particular proof step cannot be infinitely long delayed. The fairness problem of Ω-ANTS is exactly the same as in other theorem proving approaches performing depth first search. For the propositional logic fragment of our calculus for instance it is sufficient to use the automation algorithm as given in section 3.5.2 since we have to model a decision procedure. However, in the case of the first order fragment and even with higher order inference rules, we enrich the automation wrapper with an iterative deepening search in order to ensure fairness.

We can now define the notion of automation completeness.

Definition 3.13 (Automation completeness): The Ω-ANTS mechanism can be called *automation complete* with respect to a given calculus C if

(i) the agent societies specified are suggestion complete with respect to C.

(ii) the command agents for C and the suggestion agent are non-excluding.

(iii) the search procedure is fair; that is, the application of none of the suggestions is delayed infinitely long.

(iv) the resource bounds and deactivation threshold are chosen sufficiently high, such that each agent's computation terminates within these bounds.

□

Assuming that we work under the resource-abstraction assumption, our automation wrapper uses iterative deepening search and the heuristics of command and suggestion agents are non-excluding, the crucial point to show is that the single agent societies are both adequate and sufficient. We illustrate the notions of adequacy and sufficency with the example of the *AndI* agents given in table 3.2 and formalized in the preceding section.

Assertion 3.14: *The agents* $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}},\dots\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$ *are both (a) adequate and (b) sufficient to apply AndI whenever possible in automated proof search.*

Proof:

(a) To show that all computable suggestions are indeed applicable we check that each agent's lambda expression really evaluates to the desired result applied to a PAI and, in case of predicate agents, to a formula. For predicate agents the verification is performed in two steps: First we check whether the PAIs are filtered according to the agents goal, dependency, and exclude set. Then we show that given the appropriate PAI the agent actually detects formulas with the desired content and construes the correct, extended PAI.

If we examine the agent $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$, the semaphore of its first *if-then-else* statement reads

$$[Pick_{Conj}(PAI)=\epsilon]\wedge[Pick_{LConj}(PAI)=\epsilon]\wedge[Pick_{RConj}(PAI)=\epsilon]$$

which can only be satisfied by the empty PAI. We now have to check the validity of the predicate $\exists A_o\mathbf{.}\exists B_o\mathbf{.}L=[A\wedge B]$ with respect to the different possible instantiations of L. For the $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$ agent we have only two cases to consider, namely $L=p_o$ or $L=q_o\wedge r_o$, where p,q,r are arbitrary predicates and in particular p is not a conjunction. Clearly the predicate holds only for the latter with respect to which the lambda expression evaluates to the new PAI

$$[Conj{:}q_o\wedge r_o]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[]$$

which is both well constructed and of the desired form.

While the reasoning for this agent was trivial, it is slightly more complicated for agent $\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}}$. Here the analysis of the first semaphore yields that only PAIs of the form

$$[Conj{:}\epsilon]\#[LConj{:}s_o]\#[RConj{:}\epsilon]\#[]$$

are accepted, where s_o is an arbitrary proposition. This entails that the second semaphore evaluates to the form $\exists B_o\mathbf{.}L=[s_o\wedge B]$. If we now conduct a case split on L we have three cases to consider, namely $L=p_o$, $L=q_o\wedge r_o$, or $L=s_o\wedge r_o$. Again the predicate holds only for the third formula yielding as new PAI

$$[Conj{:}s_o\wedge r_o]\#[LConj{:}s_o]\#[RConj{:}\epsilon]\#[]$$

Showing the adequacy of the remaining two goal agents $\mathfrak{G}^{\{Conj\}}_{\{RConj\},\{LConj\}}$ and $\mathfrak{G}^{\{Conj\}}_{\{LConj,RConj\},\{\}}$ works analogously. All the agent considered so far have a complete specification with respect to the formal arguments of the *AndI* command; that is, the goal, dependency, and exclude sets comprise all the formal arguments of the command. Therefore, the agent is always only applicable to one type of PAI, which had to be considered when showing whether a newly constructed PAI is adequate.

For the two support agents of *AndI*, however, we have to consider several possible PAIs since they both have a degree of freedom by not explicitly specifying all formal arguments in the first *if-then-else*. Hence, we have to show for all cases of PAIs that the extended PAI is still adequate. We exemplify this for the $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ agent. Its first semaphore requires a PAI to have the *Conj* argument instantiated and the *LConj* instantiated, while there are no requirements for the *RConj* argument. If we assume that any PAI the agent is applied to is correct, we have thus two possible PAIs to consider:

1. $[Conj{:}q_o\wedge r_o]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[]$
2. $[Conj{:}q_o\wedge r_o]\#[LConj{:}\epsilon]\#[RConj{:}r_o]\#[]$

In both cases the second semaphore evaluates to $\exists B_o.[L \wedge B] = q_o \wedge r_o$. Since the existential variable B_o has to be eliminated with r_o the only possible instantiation for L is q_o in order for the predicate to evaluate to true. This leads to two possible extended PAIs:

1. $[Conj{:}q_o \wedge r_o]\#[LConj{:}q_o]\#[RConj{:}\epsilon]\#[]$
2. $[Conj{:}q_o \wedge r_o]\#[LConj{:}q_o]\#[RConj{:}r_o]\#[]$

Both PAIs are obviously correct and hence we conclude that assuming the agent $\mathfrak{G}^{\{Conj\}}_{\{RConj\},\{LConj\}}$ is applied to correct PAIs, only, it also yields correct, extended PAIs and is therefore adequate.

Since the reasoning for the second support agent $\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}}$ is analogous we can conclude that our agent society for the *AndI* command is indeed adequate.

(b) To ensure sufficiency we have to show that each PAI of *AndI* necessary for automation can be computed by cooperation of the single argument agents. Since for the first order fragment of the calculus to be complete[1] it is sufficient that all rules are applied backwards only, the possible PAIs are of the form:

(i) $[Conj{:}p_o \wedge q_o]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[]$

(ii) $[Conj{:}p_o \wedge q_o]\#[LConj{:}p_o]\#[RConj{:}\epsilon]\#[]$

(iii) $[Conj{:}p_o \wedge q_o]\#[LConj{:}\epsilon]\#[RConj{:}q_o]\#[]$

(iv) $[Conj{:}p_o \wedge q_o]\#[LConj{:}p_o]\#[RConj{:}q_o]\#[]$

Here p and q are arbitrary but fixed formulas occurring in a partial proof P. We omit to painstakingly show for all of the cases that they can be computed and instead discuss representatively case (ii). Hence we have to show that each PAI of the form $S = [Conj{:}p_o \wedge q_o]\#[LConj{:}p_o]\#[RConj{:}\epsilon]\#[]$ that is applicable in P will actually be computed. As S is applicable, P must contain an open node containing $p \wedge q$ together with a support node containing p. We also assume that $p \wedge q$ is the formula of the focused node of the active focus. Initially the command blackboard contains the empty PAI $[Conj{:}\epsilon]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[]$, to which only $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$ can be applied. Since we have already showed the adequacy of our agents we can safely reason that the agent computes $[Conj{:}p_o \wedge q_o]\#[LConj{:}\epsilon]\#[RConj{:}\epsilon]\#[]$ as the new PAI. This in turn triggers the computations of $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ and $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$. In our case we can ignore the results of the latter. Knowing that our agents are adequate and under the assumption that one support node actually contains p as formula, $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$ returns exactly the PAI in question.

When checking all other cases we can observe that for the automation mode, where pure backward reasoning is performed, the three agents $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$, $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$, and $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$ are already sufficient. ■

The last observation suggests that the other three agents are indeed needed to support user interaction, only. For instance, the user can apply Ω-ANTS to complete a particular PAI like $[Conj{:}\epsilon]\#[LConj{:}p_o]\#[RConj{:}\epsilon]\#[]$, which will trigger the computations of agent $\mathfrak{G}^{\{Conj\}}_{\{LConj\},\{RConj\}}$. Thus, the proof of assertion 3.14 provides us with valuable information for the design of the heuristics of the resource agent:

[1] BYRNES shows in [55] that the first order fragment of our ND calculus is complete, when performing only backward search with the introduction rules. He introduces a corresponding search procedure for his NIC calculus, which we discuss in more detail in chapter 5.4.1.

In order for the calculus to remain complete it is only necessary to keep the agents $\mathfrak{G}^{\{Conj\}}_{\{\},\{LConj,RConj\}}$, $\mathfrak{S}^{\{LConj\}}_{\{Conj\},\{\}}$, and $\mathfrak{S}^{\{RConj\}}_{\{Conj\},\{\}}$ always active.

3.6.3 Interaction Completeness

Interaction completeness of a calculus implies that one never has to rely on another interaction mechanism besides Ω-ANTS in order to perform possible proof steps within a given calculus. Therefore, we have to show that all possible PAIs to apply a rule interactively can be computed. This is generally a stronger requirement than for automation completeness as can be easily observed with our *AndI* example.

When automated, for instance, in the context of the NIC calculus as described in chapter 5.4.1, we want to strictly perform backward search and only the PAIs (i)—(iv) given above are thus legitimate. However, when using our calculus interactively forward reasoning is a perfectly legal option. This means that PAIs of the form $[Conj{:}\epsilon]\#[LConj{:}p_o]\#[RConj{:}q_o]\#[]$ are also legitimate. But it can be easily shown that this PAI cannot be computed with the given agent society since neither of the agents can compute an instantiation for *RConj* or *LConj* when applied to the empty PAI. Thus, we can conclude that our given society of argument agents for the *AndI* command is not interaction complete.

A second point we have to take into account for interaction completeness are PAIs preset by the user. While in automation mode the blackboards are always initialized with the empty PAI, the user can ask Ω-ANTS interactively to complete a particular PAI, such as $[Conj{:}\epsilon]\#[LConj{:}p_o]\#[RConj{:}\epsilon]\#[]$, which is then used as initial value on the blackboard. When showing interaction completeness it is necessary to show sufficiency and adequacy of the agent society for all possible initializations of the command blackboard.

3.6.4 Soundness

Soundness is not really a problem. As we presuppose that the underlying theorem proving environment takes care of a sound application of its own inference rules. Furthermore, in systems such as ΩMEGA soundness is only guaranteed at the level of primitive inferences and not necessarily for all inference rules involved. Thus, soundness requirements when computing suggestions for methods that do not necessarily lead to a correct proof would not make sense. Thus, instead of *logical soundness* we are rather interested in the notion of *applicability*. This notion relates the PAIs computed by Ω-ANTS to the particular side-conditions of the underlying inference rules (whether they are logically sound or not).

The effect of non-applicable PAIs suggested to the user or the automation wrapper might lead to failure when applying the respective command. In the current implementation such a failure will simply be ignored and the responsible PAI is discarded. However, too many non-executable suggestions might negatively influence the mechanisms user-acceptance and especially the performance of the automation wrapper.

3.7 Discussion

Since we have already motivated the choice of our particular architecture in section 3.1 we shall discuss in this section how the architecture and its distributed search compares to those already existing in the literature. In particular we shall discuss in detail how the particular design decisions of Ω-Ants help to overcome the restrictions of traditional sequential suggestion mechanisms. We shall also be interested in what exactly the form of the parallelism is that Ω-Ants implements with respect to a classification of different forms of parallelism in deduction systems. Moreover, we examine how our blackboard architecture relates to standard architectures from the literature and how our notion of an agent compares to some of the standard concepts of agent.

3.7.1 Parallel vs. Sequential

In the Ω-Ants approach to suggest commands the system steadily conducts useful computations in the background and constantly utilizes the available computational resources. Ω-Ants suggests only commands that are really applicable and offers the user the choice between several suggestions that are computed in parallel and sorted according to goal directed heuristics. This is an improvement of traditional sequential mechanisms for computing command defaults in interactive theorem provers whose shortcomings we have already discussed in section 3.1.

The decision to have the mechanism running in the background is clearly motivated by the consideration to enable user interaction even while suggestions are computed. Since the display of suggestions is constantly updated the user can also observe this process and choose one of the commands at any point of time. However, the user is not purely dependent on the suggestions alone since he can use still the full set of commands available in the theorem proving environment.

We prefer the distributed search for several reasons. The foremost is to avoid the explosion of predicates, in the worst case up to $O(n \cdot 2^{(n-1)})$ different predicates, to compute the suggestions without the sequential restriction. The cooperation of the knowledge sources, however, limits in most cases the actual number of predicates we need for computing argument suggestions.

Moreover, the distributed search leads to an anytime behavior of the mechanism in a sense that the longer it is running the better the suggestions become, at least with respect to the heuristics involved. For instance, if we would follow a naïve approach to computing suggestions in the background with a single background process, in which the suggestions are computed for one command after the other, the user would still need to wait until all possible suggestions are computed in order to get an overview of the commands, which are applicable at all. And if the user is too impatient for that it can happen that certain commands are never suggested since their applicability is never tested. On the contrary, in the distributed approach all commands have the same chance of being chosen, depending only on whether they are actually applicable in the proof state and how much time the computations for their suggestion takes. Finally, the distribution also increases the robustness of the suggestions since any error in a single thread might lead to missing suggestions but does not lead to a failure of the overall mechanism. Moreover, we can embed uncertain components, such as undecidable procedures, without putting the overall mechanism at risk.

The reason for choosing a two layered architecture is the nature of the knowledge gathering process, which proceeds in two steps: First we compute knowledge about which arguments can be instantiated for each command and from that we can compile knowledge about which of the commands can be actually applied. The process is also reflected in the centralized structure of blackboard architecture. The knowledge about the proof and which commands are applicable is compiled to help the user during interactive theorem proving. We also want to be able to use heuristics in order to decide which of the suggested commands is the best in a given proof state by considering all suggestion computed up to a certain point, which is easier when all necessary information is available on a blackboard.

3.7.2 Parallelism of Deduction

In [36, 37] Bonacina identifies three different notions of parallelism in the context of deduction. Although her considerations are based on refutation based first order theorem proving we adapt her notions for our context. In detail the types of parallelism are the following:

Parallelism on the term level: This means subexpressions can be accessed and subtasks of inferences can be performed in parallel.

Parallelism on the clause level: This is basically parallelism at the inference level; that is, several inferences are done in one step.

Parallelism on the search level: Thereby multiple deductive processes search in parallel.

If we classify the parallelism realized in Ω-Ants with respect to this taxonomy we have basically modeled all three forms of parallelism. Parallelism on the term level is clearly realized since our argument agents can access sub-terms in parallel during their search. Moreover, they can also compute things like term instantiations and matchers etc. and thereby perform subtasks of inferences in parallel.

Since Bonacina gives her taxonomy for first order refutation procedures she is mainly concerned about how inferences on clauses are performed in parallel. In this context she considers things such as hyper-resolution rules or parallel term rewriting steps. Thus, parallelism at the inference level corresponds in a loose sense to tactics in our context that perform a series of inference steps. (E.g., consider the tactic $\equiv_E^*$ we shall introduce in chapter 5.4.3, which performs several expansions of a defined concept in one step.) Hence, this type of parallelism is not really directly connected to Ω-Ants but already realized in Ωmega's overall concept.

The last point, the parallelism on the search level, is given by the possibilities of integrating automated reasoning procedures into argument agents as explained in section 3.5.1 as well as the close interlink between automated and interactive search procedures given in section 3.5.3, where subproblems in non-active foci can be solved in the background.

Apart from the above taxonomy there are also other possible classification criteria, for instance, to identify *and-or parallelism* (see [38]) or *cooperation* and *competition* of parallel components (see [199]).

And parallelism means that several subgoals can be treated in parallel. In Ω-Ants this type of parallelism is realized in the distribution as background processes as presented in section 3.5.3. In contrast, *or* parallelism means that several alternative proofs for one subgoal are constructed in parallel. So far, *or* parallelism can only be realized partially in Ω-Ants since automated theorem provers can search for a proof of the same subgoal in parallel and if several have been found one of the proofs can then be selected. However, the $\mathcal{PDS}$ itself does not allow for constructing and storing several alternative proofs. An appropriate expansion of the $\mathcal{PDS}$ will be subject of future work.

The division into cooperating and competing parallel components is essentially an aspect of the parallelism on the search level. We have deductive processes that try to solve a common goal either jointly or concurrently. In the Ω-Ants mechanism both approaches can be modeled. On the one hand, we have several automated theorem provers attached to different commands that try to solve a given goal concurrently. On the other hand, some automated theorem provers can return partial results that can then in turn be tackled by other provers. This latter type of cooperation in Ω-Ants has been successfully used to automatically solve a class of examples in set theory, where the higher order resolution prover Leo has been used to simplify given higher order problems to a level where either the first order resolution prover Otter could take over to finish the proof or the model generator SatchMo was able to conclude that the given problem was not a theorem. For a detailed account of these experiments see [23, 24].

3.7.3 Blackboard Architecture

The Ω-Ants architecture has similarities to several of the classical blackboard architectures as for instance discussed in [79]. Since most blackboard architectures are descendants from either the HearsayII [80] or the Hasp [161] architecture we mainly consider those two in our discussion. Both classic architectures consist of a single blackboard, which is, however, hierarchically structured in itself. Their knowledge sources can then either work one of the hierarchies or propagate from a lower to an upper level thereby possibly omitting intermediate levels. Both knowledge sources and entries on the blackboard can be very heterogeneous.

In Ω-Ants we have modeled hierarchies by using a two layered architecture of blackboards. This leads on the one hand to less flexibility for the knowledge sources since our agents can all only work exactly for one level in the hierarchy. On the other hand, the setup enables us not only to separate horizontally into several hierarchies, but also vertically by clustering the agents on the lower level into societies working for separate command blackboards. Moreover, our blackboards are rather homogeneous with respect to both, their entries and their knowledge sources.

The composition of our argument agents is similar to the knowledge sources of the HearsayII blackboard, which are condition-action pairs. We can also view an argument agent as consisting of a condition, given by the necessary composure of a blackboard entry, and an action, the search or computation it performs. In contrast, both command and suggestion agents work with respect to a set of heuristics and are thus comparable with the knowledge sources of the Hasp blackboard, which are sets of rules.

Our classifying agents and the resource agent do not really fit into this picture of knowledge sources since their information is propagated downward in the architectures. This is in contrast to the direction of data-flow on the HearsayII and Hasp blackboards, which is upward in the hierarchies, which is also the main direction of data-flow in Ω-Ants. However, if we look at how the computation of knowledge sources is triggered we can detect similarities for classifying and resource agents as well.

On the HearsayII blackboard the knowledge sources are data-driven; that is, they start their action as soon as an appropriate blackboard entry meets their condition. This is also the way our suggestion, command, and argument agents in Ω-Ants act. In the Hasp architecture, however, knowledge sources are control-driven; that is, knowledge sources on a lower level are triggered directly and exclusively by knowledge sources on a higher level. This is comparable to the influence of the resource and classifying agents, which trigger from the top level changes of computational behavior of the agent societies on the lower levels. This control is, however, dynamic in contrast to the Hasp blackboard, where the control is directly implemented into the knowledge sources.

Since both HEARSAYII and HASP are single blackboard architectures without parallelism we have to compare the distribution and concurrency aspects of Ω-ANTS to another blackboard architecture. Here the POLIGON [178] blackboard architecture is the one that matches most closely our approach. However, in this architecture the single blackboards are more loosely connected than in the strict hierarchical structure of Ω-ANTS. POLIGON resembles thus more a modern multi-agent architecture than a classical blackboard approach. One of the major problems the POLIGON architecture deals with is the scheduling problem; that is, when knowledge sources work in parallel how can we avoid that one knowledge source destroys a blackboard entry that is the working bases of another knowledge source. This is solved by knowledge sources putting locks on the entry they are currently working on. This phenomenon does of course not occur with the parallel Ω-ANTS agents since they always only write new extended entries on the blackboard without modifying old entries.

The discussed blackboard architectures permit to measure the performance of the knowledge sources. This data is measured and collected centrally by the blackboard and exploited in subsequent scheduling processes. The knowledge sources themselves have no means of measuring their performance and validating their effectivity. While the central approach to evaluate performance data and influence the scheduling is comparable to the reasoning of our resource agent and use of the activation/deactivation threshold. Differing is however that our argument agents can measure and evaluate their own performance and possibly change their state of activity by rewarding or penalizing themselves.

3.7.4 Knowledge Sources vs. Agents

Ω-ANTS is a blackboard architecture as opposed to a real multi-agent system. However, we call the knowledge sources of our blackboards agents as they have certain properties that distinguish our agents from common knowledge sources in the traditional blackboard architectures. So, to what extend do our knowledge sources qualify as agents?

For the definition of the notion of an agent we best start with a remark by NWANA and NDUMU given in [163]:

> We have as much chance on agreeing on a consensus definition for the word 'agent' as artificial intelligence researchers have of arriving at one for 'artificial intelligence'.

Similarly other authors also concede that there is no universally accepted definition for the term agent (cf. [210]). However, there is a certain concensus on at least some of the attributes a computational entity has to exhibit in order to be called an agent.

In the prologue of [210] WEISS gives some criteria for "interacting, intelligent agents" while admitting that these are only explanations for "...what is generally considered to be essential for an entity to be an intelligent agent" and not part of a universally accepted definition [211]. WEISS considers agents as autonomous, computational entities that perceive their environment and act upon it. They are intelligent in a sense that they perform their task in a certain goal-directed manner in order to optimize their own performance. To pursue their goal they have to operate flexibly and rationally in given situations. This is achieved by deliberative abilities, such as reasoning on internal states or some representation of the environment, but also with interaction capabilities to either cooperate or compete with other agents.

In the introduction of [210] Wooldridge specifies similar properties for intelligent agents [214]. For Wooldridge the essential property is the capability of flexible autonomous actions, which he characterizes with three abilities: *reactivity*, *pro-activeness* and *social ability*. Reactivity means that agents are robust in the sense that they can adapt to the changes in their environment, while pro-active agents exhibit not only goal-directed behavior but also take the initiative to pursue their goals. Finally the social abilities enable agents to negotiate with other agents to share goals and cooperate. Like Weiss, Wooldridge emphasizes that his definition of an intelligent agent is by no means a universally accepted one.

If we compare our agents and their abilities to these notions of autonomous intelligent agents we can see our agents are only autonomous from a software engineering point of view since they are implemented in concurrent threads. From an architectural point of view they are only meaningful in the context of the blackboard or blackboards they work for. Although the agents cooperate to compute suggestions they do not exhibit any real social abilities, in the sense that they can dynamically decide with whom to cooperate or which society to join. Instead the societies are predetermined by the agents' specification. The agents are, however, proactive in a sense that they are not explicitly triggered or scheduled by the respective blackboard. In fact, the blackboards do not even have an overview which agents they have as knowledge sources, since our agents commence their own tasks by picking up suitable information from the blackboards. Our agents are also partially adaptive. On the one hand they adapt their search with respect to given PAIs and they react to the knowledge and resource information given on the blackboard.

Overall our agents show a reactive behavior and are also very robust. We can, however, attribute them a few abilities from the definition of a deliberative agent. The argument and command agents have some knowledge on their capabilities, which is a priori implemented into them, and they gather dynamically information on their performance. This gives them a certain internal state, which in turn is used to react to changes on the blackboards in the from of given information or resource criteria. However, command, argument, suggestion and classifying agents have no explicit representation of their environment. Only the resource agent has a certain representation of the status of the overall system and reasons accordingly. Our agents also have no real planning capabilities, although some agents can integrate complex inference procedures. But these procedures are merely their program and not something they can use to reason about their own internal state and their effects on the environment.

To conclude our discussion we can consider our agents as simple reactive agents communicating via blackboards. The complete Ω-Ants architecture can then be seen as a distributed problem solving system instead of a multi-agent system following [39]. But as pointed out by Weiss the modern concept of multi-agent systems covers both types of systems and makes therefore such a distinction obsolete [211].

3.8 Summary of Chapter 3

In this chapter we have introduced Ω-Ants, a distributed architecture to support both interactive and automated theorem proving in Ωmega. It is based on a two-layered blackboard architecture, which reflects the knowledge gathering process to detect applicable commands for inference rules. On the lower layer possible instantiations for formal arguments of the commands are searched, on the upper layer all in a proof situation applicable commands are collected. These can then either be applied by the user interactively or automatically by an automation wrapper for Ω-Ants.

The Ω-Ants mechanism has the ability to dynamically adapt itself to a given problem or sub-problem by gathering additional information on the proof in order to narrow its search. It also has a resource-adaptive behavior in order to search more efficiently. Moreover, their are various heuristics that can be changed to interactively adapt Ω-Ants behavior.

Ω-Ants overcomes the limits of traditional mechanisms to suggest commands or default parameters for possible argument instantiations. Its distributed architecture with its concurrent computation as well as the resource-adaptivity allow for an easy integration of external reasoners and gives thus a basis for cooperation of various automated components. Together with the automation wrapper it can also be used as a parameterizable automated theorem prover that still allows for user interaction. And since most of its components can be formalized in a regular lambda calculus we can also undertake theoretical considerations with respect to an adequate modeling of an underlying calculus as well as to gain valuable knowledge for the design of adequate heuristics.

Chapter 4

Integration of Reasoning Techniques

In this chapter we shall investigate different ways to combine reasoning techniques within a proof planning system. In particular we shall examine how the top-down reasoning approach of a proof planner can be combined with the bottom-up behavior of Ω-Ants and how symbolic computations can be soundly integrated into proof planning.

In the first part of this chapter we investigate combinations of the Ω-Ants mechanism with Ωmega's multi-strategy proof planner Multi. Such a combination enriches the traditional sequential-style proof planning with aspects of parallelism and concurrency. Our investigations so far have concentrated on two aspects of proof planning where parallelism is particular useful, namely in interactive proof planning and to compute applicable assertions during automatic proof planning.

In the second part of this chapter we present a technique for the sound integration of computer algebra systems into proof planning. It is based on the idea to separate computation and verification and can thereby exploit the fact that many elaborate symbolic computations are trivial to verify. In proof planning the separation is realized by using a powerful computer algebra system during the planning process to do non-trivial symbolic computations. Results of these computations are checked during the refinement of a proof plan to a calculus level proof using a small, self-tailored system that gives us intermediate information on its calculation. This information can be easily expanded into a checkable low-level calculus proof ensuring the correctness of the computation.

4.1 Combining Ω-Ants and Proof Planning

In this section we examine ways of combining the Ω-Ants architecture with the multi-strategy proof planner Multi [155]. The main motivation for such a combination is twofold. Firstly, both systems have contrary reasoning approaches. While proof planning is essentially top-down reasoning (i.e., both methods and applicability tests are on a high level), Ω-Ants's search behavior is bottom-up in the sense that small pieces of information on the proof are assembled to determine possible inference steps. The integration of these contrary reasoning approaches can strengthen an overall reasoning system. For instance, gaps in a proof plan that are not covered by one of the planning methods can be closed using a bottom-up search of Ω-Ants. A second motivation for the combination is that proof planning, as implemented in MULTI, is essentially a sequential approach to reasoning, while Ω-ANTS tries to parallelize the reasoning process as much as possible. However, certain parallel features might improve performance and power of the proof planner.

The benefits of a full integration of proof planning and Ω-ANTS are not only that the proof planner can gain interesting and valuable parallel features but also that Ω-ANTS gains access to MULTI's strategies and methods to combine them with its own distributed search technique. While the full-fledged integration of MULTI and Ω-ANTS has still to be investigated in the future, in this thesis, we are already dealing with two aspects of enriching MULTI with Ω-ANTS's parallel search behavior:

1. To enable interactive proof planning with MULTI.
2. To compute applicable assertions during automatic proof planning.

(1) was motivated by the need to have an interactive planning mode available, which can in particular be used when ΩMEGA serves as the backend of a tutor system such as [154]. Ω-ANTS is defined as a search algorithm for MULTI and can then be parameterized with appropriate planning strategies. The applicability of single methods is then checked by Ω-ANTS-agents and applicable methods are suggested to the user similar to the regular proof rules that are suggested during traditional interactive theorem proving. And instead of using the control rules to guide the search this is done by Ω-ANTS sorting heuristics.

(2) is an application in which the concurrency of the Ω-ANTS mechanism is fruitfully exploited. Thereby Ω-ANTS is used to retrieve applicable mathematical theorems during automatic proof planning. This frees the actual, sequential proof planning algorithm from computationally expensive test of applicability for single theorems. The theorems of the knowledge base are automatically divided into different classes of theorems where each is assigned to a blackboard. The single theorems of the respective classes are checked for applicability in parallel, possibly using different criteria to decide applicability in a given proof context. Applicable theorems are gathered on the blackboards and suggested to the proof planner, which in turn exploits this information during the proof planning process.

4.1.1 Using Ω-ANTS as Algorithm in MULTI

For interactive planning we want the user to be able to choose from all methods that are applicable in one planning state. We also want to avoid the effect of control rules suppressing methods in certain proof states since the user might have a planning approach that is different to the automatic one. Therefore, in each planning state we want to compute all applicable methods and present them to the user possibly ordered with respect to some sorting criteria.

This cannot be done with MULTI's regular planning algorithm *PPlanner* without making major modifications to the algorithm itself: Firstly, because *PPlanner* follows a depth first search approach; that is, in each planning step the available methods are sequentially checked for applicability and if an applicable method has been found it is applied immediately. All remaining methods are checked for applicability only in case the planner ever backtracks to this planning state. Secondly, the available methods are sometimes structured or restricted by control rules that are optimized for automatic proof construction.

Thus we use Ω-ANTS as an algorithm in MULTI and define agents to test for the applicability of single methods. This has not only the advantages that the

Strategy: `homomorphisms-interactive`			
Condition	HomomorphismProblem		
Action	Algorithm	Ω-Ants	
	Parameters	Methods	$HomOnDomain$, $Homomorphism$, $ElemOfKernel$, $\forall_I Sort$, $\exists_I Sort$, ...
		Agents	$\mathfrak{C}_{HomOnDomain}$=$\{\mathfrak{S}^{\{L_1\}}_{\{\},\{\}}, \mathfrak{G}^{\{L_3\}}_{\{L_1\},\{\}}\}$, ...
		Heuristics	$PropComplete$, $LeastSubGoals$

Figure 4.1: The interactive strategy `homomorphisms-interactive`.

applicability of the methods can be checked in parallel but also that applicable methods can be suggested to the user in a sorted way using Ω-Ants's heuristics.

4.1.1.1 Interactive Strategies

In Multi an algorithm is a means to modify partial proof plans. It has a set of parameters to influence its behavior. A strategy is then the concrete parameterization of the algorithm. When employing Ω-Ants as an algorithm of Multi its parameters are the available methods, the argument agents corresponding to these methods and the heuristics for the command agents and the suggestion agent.

Figure 4.1 depicts the `homomorphisms-interactive` strategy that is used to interactively plan homomorphism proofs as described in more detail in part III of this thesis. We can observe that the structure of this strategy is very similar to the structure of the `TryAndError` strategy given in chapter 2.2.4 figure 2.5. The only difference is that instead of slots for control rules and termination we have slots for agents and heuristics. The elements of the agents slot of the strategy are here given in the form of sets of argument agent societies associated with the corresponding command agent. In order to preserve space in figure 4.1 we have given only one example of an agent society for the *HomOnDomain* method.

The first of the given heuristics in our example strategy ensures that only methods are suggested to the user that are actually applicable in the given planning state. It is a sorting heuristic for the command agents that in particular allows them to propagate only those PAIs to the suggestion blackboard that correspond to complete matching of the associated method. The second heuristic is a sorting criteria for the suggestion agent, which states that methods are preferred that generate the least new open subgoals. This sorting criteria does of course not necessarily lead to the best possible suggestions.

4.1.1.2 Determining Applicable Methods

Method applicability is determined by Ω-Ants similar to regular commands by computing PAIs containing the necessary arguments to apply a method in a given proof state. But unlike to commands the argument pattern of a method cannot only consists of formal arguments for (1) proof lines and (2) additional parameters, but also contains formal arguments for (3) application conditions.

(1) are the proof lines that have to be given in the proof in order for the method to be applicable. In particular, they are the elements of the premises and conclusions slot of the method that either are unsigned or have a ⊖ sign. They are computed by predicate agents, which use directly the specification in the declarative content in order to search for matching lines. The proof lines given as actual arguments in the PAI are then the lines the method can actually be applied to.

Method: *HomOnDomain*		
Premises	$L_1, \oplus L_2$	
Appl. Cond.	$[a_1 \in A]$ & $[a_2 \in A]$	
Conclusions	$\ominus L_5$	
Declarative Content	$(L_1)\quad \Delta\quad \vdash Hom(f,(A,\circ),(B,\diamond))$	
	$(L_2)\quad \Delta\quad \vdash \Phi[f(a_1 \circ a_2)]$	$(Open)$
	$(L_3)\quad \Delta\quad \vdash \Phi[f(a_1) \diamond f(a_2)]$	$(ApplyHom\; L_1\; L_2)$

Figure 4.2: The *HomOnDomain* method.

(2) The additional parameters are just like regular parameters of commands. They are computed by function agents or, if possible, as side effect of the computations of predicate agents. The parameters given in a PAI are passed to a method as additional parameters when it is applied.

(3) The additional application conditions of a method that have to hold for the method to be applicable can be checked in two ways: Either they are implicitly checked within the search predicate of some predicate agent. Or they are explicitly checked with additional function agents. For the latter case we have to introduce dummy abstract arguments into the respective PAIs; that is, the actual instantiations of these abstract arguments are not actually required by the method when it is applied. Nevertheless, these parameters are important, since their instantiation determines the applicability of the method. If an application condition holds the respective formal argument is assigned the result as actual argument. If it does not hold the formal argument is assigned the empty actual argument ϵ. The application conditions of a method can be checked using either only one function agent or several for different conditions. In particular, we can check disjunctive application conditions of a method with several parallel agents [30].

The sorting of the entries on the command blackboard and propagation of the applicable methods to the suggestion blackboard is controlled by heuristics. Normally the heuristic of the command agents ensures that the most complete entries are sorted to the top and that methods are only reported as applicable to the suggestion blackboard when the topmost PAI on the command blackboard is complete. This corresponds to the *PropComplete* heuristic of the `homomorphisms-interactive` strategy and it has the effect that only methods are suggested that are for sure applicable in the current proof state. Since still non-instantiated arguments in the PAI can mean that some required proof lines or parameters could not be computed or that some application conditions failed.

This condition can naturally be relaxed such that methods are suggested to the user where either some arguments have to be provided manually before they can be applied or that are not at all applicable in the given proof state. This has practical use, for instance, when the system is used for tutoring purposes [154]. However, then the designer of the agents should make sure that the arguments that are computed by the agents and that possibly need to be supplied by the user should be meaningful and thus try to eliminate as far as possible dummy arguments.

Our concrete example is the *HomOnDomain* method given in figure 4.2. Its task is to apply backwards the homomorphism given in line L_1. Here line L_1 does not have the required form of its justification explicitly taken, instead it is left void. This indicates that L_1 can have an arbitrary justification. The homomorphism in line L_1 is of the form $f : (A, \circ) \to (B, \diamond)$ and is applied to a goal line containing an application of the operation $\diamond$. This is indicated in line L_3 of the method by the schematic formula $\Phi[f(a_1) \diamond f(a_2)]$, where Φ is an arbitrary proposition containing a sub-term of the form $f(a_1) \diamond f(a_2)$. The additional application condition specifies that a_1 and a_2 are actual elements of the domain A of the homomorphism f. When the method is applied the occurrence of $f(a_1) \diamond f(a_2)$ in Φ is replaced by $f(a_1 \circ a_2)$.

As indicated by the `homomorphisms-interactive` strategy there are two agents to determine the applicability of the method. The first, $\mathfrak{S}^{\{L_1\}}_{\{\},\{\}}$, searches for lines containing a definition of an homomorphism in the proof plan; that is, the agent looks for a line matching the specification of the formula of L_1 in the method's declarative content. Once it has found an appropriate line the second agent, $\mathfrak{S}^{\{L_3\}}_{\{L_1\},\{\}}$ looks if there is an open subgoal matching line L_3 of the method. Thereby it uses the matching for L_1 to look for applications of the operation of the homomorphism's domain in open goals. In case it has found such a line it additionally checks the application conditions of the method making sure that the elements in question actually belong to the domain of the homomorphism. Since the instantiation of the argument patterns of methods are more rigid than those of tactics the number of agents that need to be specified for methods are generally relatively small. For instance, line L_2 of the *HomOnDomain* method is always introduced during the method's application and thus we do not have to specify an agent looking for an appropriate instantiation in the partial proof.

4.1.1.3 Interactive Proof Planning

During interactive proof planning the user has essentially three different means of interaction: Applying methods, choosing meta-variable instantiations, and backtracking. There is also the possibility to automate the application of some methods before regaining the interactive control.

Method Application Similar to choosing regular commands methods are applied when the user chooses them interactively from those given on the suggestion blackboard. Thus, the combination of the methods is according to the users choices since control rules that otherwise would influence the planners behavior are not in effect. The user can also choose which of the possible goals will be considered for planning, which corresponds to the regular shifting of the active focus.

On the contrary, the application of normalization and restriction methods is done automatically and exhaustively after each step. This is different from the regular planning algorithm *PPlanner* where the normalization and restriction methods are tested for applicability after each regular planning step, but their actual application can be influenced by control rules (see chapter 2.2.4). The applicability of these methods is not checked with agents but with MULTI's regular matching algorithm for methods.

Meta-variable Instantiation Besides the applicable methods the user also gets a display of the meta-variables that are not yet instantiated. The instantiation can then be achieved in three different ways: (1) A meta-variable is instantiated during a method application, (2) the user provides interactively an instantiation, or (3) an agent computes and suggests an instantiation. The latter case is especially useful if a certain meta-variable is normally instantiated by a control rule in the *PPlanner* algorithm. Since in the interactive mode this particular control rule is not available it can be modeled with an agent that then suggests the computed instantiation. Its application however is still subject to the users consent.

Backtracking The backtracking on the other hand is also done interactively. Thereby the user is presented with a choice of possible backtracking points that actually correspond to the available strategies of the *BackTrack* algorithm. For instance, the user can backtrack the last planning step, the last interactive meta-variable instantiation, or also the whole interactive planning strategy and start anew with a different strategy.

Automation Naturally the application of methods with Ω-Ants can be automated with the regular automation wrapper presented in chapter 3.5. But, since the interactive strategies in Multi are rarely constructed on their own but are rather a supplement to the corresponding *PPlanner* strategy, there is also the possibility to switch from the interactive application of methods to the automatic application with Multi and vice versa. This is particularly useful for tutoring purposes and for debugging strategies. Another advantage of the Ω-Ants algorithm is that we can also use regular commands for rule, tactics or external reasoners to be intermixed with application of planning methods. This can however mess up Multi's backtrack algorithms.

4.1.2 Using Ω-Ants for Assertion Applications

In the preceding section we have seen an application of Ω-Ants within proof planning that mainly exploited its support for user interaction and its parallel features. In this section we now discuss how we can use Ω-Ants to determine the applicability of *assertions* (i.e., axioms and theorems) during automatic proof planning. This particularly exploits the concurrency of Ω-Ants in order to not only parallelize the search for applicable assertions but also to separate it from the sequential proof planning algorithm.

4.1.2.1 Assertion Applications

Working directly with assertion applications gives a more abstract layer of reasoning than the basic calculus level. In fact, Huang has identified the *assertion level* as a well defined abstraction level for natural deduction proofs [116, 117].

To clarify the notion of assertion application we pick one of Huang's examples as given in [117]. An assertion application is for instance the application of the theorem *SubsetProperty* of the form

$$\forall S_1 \centerdot \forall S_2 \centerdot S_1 \subset S_2 \equiv \forall x \centerdot x \in S_1 \Rightarrow x \in S_2$$

in the following way:

$$\frac{a \in U \quad U \subset F}{a \in F}\ Assertion(SubsetProperty)$$

The direct application of the assertion is thus an abbreviation for a more detailed reasoning process involving the explicit derivation of the goal $a \in F$ from the two premises by appropriately instantiating and splitting the *SubsetProperty* theorem.

In Ωmega assertions are applied using a specialized *Assertion* tactic. Its purpose is to derive a given goal from a set of premises with respect to a theorem or axiom. It thus enables a more abstract reasoning with respect to given assumptions. We can depict the assertion tactic as a general inference rule in the following way

$$\frac{Prems}{Goal}\ Assertion(Thm)$$

where *Prems* is a list of premise nodes, *Goal* is the goal to be proved and parameter *Thm* is the assertion that is applied.

Traditionally when proof planning with Multi assertions are applied using the *ApplyAss* method and the `select-theorems` control rule. *ApplyAss* is a generic method for assertion application and `select-theorems` is a control rule that can be parameterized with the single theorems that should be considered as applicable or with the name of theories from which all assertions should be considered[1]. The applicability of assertions is checked sequentially in the application conditions of the *ApplyAss* method by matching each given assertion with the current goal. The methods applicability in turn is checked as usual during the proof planning process. The *ApplyAss* method is equipped with a first order matching algorithm with α-equality on λ-abstractions, which might not always be sufficient for more complicated theorems. However, full higher order matching has to be avoided in order to keep method application decidable. Thus, for certain theorems or classes of theorems whose applicability cannot be checked with first order matching but maybe with other special decidable algorithms special methods can be implemented.

This way of applying assertions during proof planning has several drawbacks. Firstly, the check for applicable theorems could be parallelized to gain efficiency. Moreover, the applicability of an assertion does only depend on the given goal to be proved and not on additional information of the planner or the method. Therefore, it can be easily decoupled from the regular process of checking method applicability. A second defect is that theorems are explicitly referred to in the control rule either by their own names or the names of the theory they belong to. This means that the planner not only needs direct knowledge on the status of the knowledge base, but also, since the reference to the theorems or theories is by name, that any renamings in the knowledge base can destroy the planners behavior. Likewise, if new theorems are added they either have to be explicitly added to the control rule or they are added automatically via the theory they belong to, no matter if they are relevant for a given strategy or not. And since the control rule only is executed once during a planning process, when the respective strategy is selected, there is no possibility to dynamically add theorems to be considered. A last drawback is that if we want to incorporate more elaborate or more specialized algorithms to perform assertion matching then the regular first order matching algorithm we have to implement special methods instead of reusing the generic *ApplyAss* method.

4.1.2.2 Finding Applicable Assertions

In particular the dilemma of sequential testing suggests to employ the Ω-ANTS mechanism to search for applicable theorems. Its use can also push the search into the background and thereby decouple the computation of a possible assertion application from the actual planning algorithm.

Its main idea is to form clusters of theorems by grouping them with respect to an additional specification, such that theorems are selected that comply to a given predicate. For instance, we can form a cluster of theorems that are all concerned with a particular property. Then we can preselect according to the given goal whether the theorems of a given cluster could be successfully matched without having to carry out all possible matchings. Selecting the theorems via specifications has the advantage that the respective strategy only has to specify which type of theorems should be considered and these are automatically filtered from all those available. This does not only enable a more refined selection of theorems but also makes us independent of choosing particular theories or name references. Furthermore, new theorems can be dynamically added and fitted into the existing clusters.

All clusters are of a similar composition. In detail, each cluster is associated

[1] Naturally, in order to keep assertion application feasible it has to be restricted to a certain number of theorems and axioms that are to be considered. These assertions are selected with respect to the strategy and domain.

with exactly one command blackboard. The PAIs communicated on the blackboard correspond to the three parameters an assertion application needs, namely goal line, assertion and a list of possible premises of the assertion. The society of argument agents working for the blackboard is composed of one goal agent, one support agent and one or several function agents, which have the following tasks:

- The goal agent checks whether the cluster can possibly suggest the application of one of its theorems with respect to the given goal.
- The function agents are responsible for a certain set of theorems. They possess a predicate with which they can check whether a theorem out of their set is applicable to a given goal line using either regular matching or some special algorithm. Furthermore, they have an additional *acquisition predicate* that enables them to determine whether a theorem from the knowledge base fits into the cluster.
- The support agent in turn seeks lines in the proof that can be used as premises of a successfully matched theorem.

Thus, all clusters are of uniform setup except for the number of function agents. Here we allow for the cluster to have several function agents such that each can try to match a different set of theorems with a different algorithm. However, we do not require two sets of theorems of two different function agents to be necessarily disjoint.

Similar to their uniform composition, clusters also function similarly to determine applicability of theorems. First the goal agent determines whether the focused node of the active focus contains a formula complying to the agent's predicate. If the test is affirmative the agent writes the goal in a new PAI on the blackboard. Then the single function agents start working in parallel and try to match their theorems with the given goal. Thereby the theorems are chosen with respect to the theory of the problem; that is, only those theorems are matched that either belong to the problem's theory or one of the inherited theories. All other problems are not checked since they were not applicable anyway. For each theorem that matches successfully they add a new PAI to the blackboard containing both the goal and the theorem. These PAIs in turn trigger the support agent, which then takes the matched theorem, extracts the necessary information on which premises are required for its application and searches for suitable lines in the support nodes of the active focus. In case it is successful it adds a list of support nodes to the respective PAI. The support agent has a uniform implementation for all clusters.

Each command agent surveying the command blackboard of a theorem cluster sorts the entries on its blackboard with respect to the completeness of the PAIs. It passes all those entries to the suggestion blackboard containing a matched theorem and updates them if necessary. The only task of the suggestion agent is to signal the availability of theorem suggestions to the planner. If the planner requests these suggestions, the suggestion agent passes all available theorem suggestions unsorted to the planner. The idea of not sorting the suggested theorems is to leave the decision to the planner to choose from all applicable theorems using its meta-knowledge on both the current problem and the state of the planning process.

The planner initializes the mechanism only once when it first wants to use it. From then on the computations are triggered automatically whenever a change in the proof occurs and a new open goal is created. In case the planner wants to explicitly exclude the applications of assertions during the planning process it can suspend the mechanism and later resume it without explicitly reinitializing it.

This procedure has the advantage that the original construction of clusters of assertions has to be done exactly once during the actual initialization. In this phase the function agents construct the theorem clusters by using their acquisition predicate on all theorems available so far in Ωmega and either acquire them for their set of theorems or reject them. Whenever new theorems become available — for instance, by loading an additional theory from the knowledge base or by adding a newly proved problem — this is detected by a classifying agent, which subsequently passes these new theorems as information into the mechanism. The function agents react to this information by checking whether the new theorems fit into their cluster with respect to the acquisition predicate. In case a new cluster is added, for instance by the planner, it is regularly initialized; that is, it goes through the complete process of initial theorem acquisition.

4.1.2.3 Example

Our example is taken from the case study on the proofs of properties of residue classes presented in chapter 7.3.3. Since we shall discuss this example in great detail there we are not concerned with details of the formalization and the proof here. Instead we concentrate on how Ω-Ants determines the applicability of assertions in the case of the example.

We consider the first step in the proof of the theorem

$$Conc.\ \vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare (x \bar{*} y) \bar{+} \bar{3}_5).$$

It states that the given residue class set $\mathbb{Z}_5$ is closed with respect to the operation $\lambda x_\blacksquare \lambda y_\blacksquare (x \bar{*} y) \bar{+} \bar{3}_5)$. Here $\bar{3}_5$ is the equivalence class of all integers that are congruent to 3 modulo 5 and the dashed operations are the obvious operations on the equivalence classes modulo 5. The complete proof of the theorem is given in table 7.4 in chapter 7.3.3.

Among the theorems we have for the domain of residue classes there are some that are concerned with statements on the closure property. In particular, we have the following six theorems:

$$\begin{array}{ll}
ClosedConst: & \forall n{:}\mathbb{Z}_\blacksquare \forall c{:}\mathbb{Z}_{n\blacksquare} Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare c) \\
ClosedFV: & \forall n{:}\mathbb{Z}_\blacksquare Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare x) \\
ClosedSV: & \forall n{:}\mathbb{Z}_\blacksquare Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare y) \\
ClComp\bar{+}: & \forall n{:}\mathbb{Z}_\blacksquare \forall op_{1\blacksquare} \forall op_{2\blacksquare} (Closed(\mathbb{Z}_n, op_1) \land Closed(\mathbb{Z}_n, op_2)) \Rightarrow \\
 & \qquad Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare (x\ op_1\ y) \bar{+} (x\ op_2\ y)) \\
ClComp\bar{-}: & \forall n{:}\mathbb{Z}_\blacksquare \forall op_{1\blacksquare} \forall op_{2\blacksquare} (Closed(\mathbb{Z}_n, op_1) \land Closed(\mathbb{Z}_n, op_2)) \Rightarrow \\
 & \qquad Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare (x\ op_1\ y) \bar{-} (x\ op_2\ y)) \\
ClComp\bar{*}: & \forall n{:}\mathbb{Z}_\blacksquare \forall op_{1\blacksquare} \forall op_{2\blacksquare} (Closed(\mathbb{Z}_n, op_1) \land Closed(\mathbb{Z}_n, op_2)) \Rightarrow \\
 & \qquad Closed(\mathbb{Z}_n, \lambda x_\blacksquare \lambda y_\blacksquare (x\ op_1\ y) \bar{*} (x\ op_2\ y))
\end{array}$$

The theorems *ClosedConst*, *ClosedFV*, and *ClosedSV* talk about residue class sets with simple operations whereas $ClComp\bar{+}$, $ClComp\bar{-}$, and $ClComp\bar{*}$ are concerned with combinations of complex operations. Therefore, the difference between the groups of theorems is that the applicability of former can be checked with first order matching whereas for the latter we need higher order matching. For example, when applying the theorem $ClComp\bar{+}$ to our problem at hand the necessary instantiations for the operations have to be $op_1 = \lambda x_\blacksquare \lambda y_\blacksquare x \bar{*} y$ and $op_2 = \lambda x_\blacksquare \lambda y_\blacksquare \bar{3}_5$, which cannot be found by first order matching. However, since we are concerned with only a distinct set of binary operations and their combinations, we can keep things decidable by using a special efficient algorithm, which matches the statements of the theorems $ClComp\bar{+}$, $ClComp\bar{-}$, and $ClComp\bar{*}$ with nested operations on congruence classes.

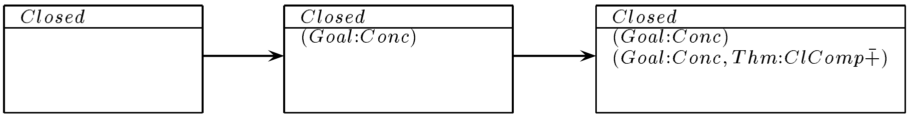
In Ω-Ants we have the agent society as depicted in figure 4.3 for the cluster comprising the theorems given above. The first agent is the goal agent that accepts

$$
\begin{array}{lcl}
\mathfrak{G}^{\{Goal\}}_{\{\},\{Thm,Prem\}} & = & \left\{Goal\text{: } Goal \text{ contains the } Closed \text{ predicate}\right\} \\
\mathfrak{F}^{\{Thm\}}_{\{Goal\},\{Prem\}} & = & \left\{Thm\text{: Conclusion matches } Goal \text{ with first order matching}\right\} \\
 & & \left\{\begin{array}{ll} Acquisition\text{:} & \text{Conclusion contains } Closed \text{ as outermost} \\ & \text{predicate and a simple operation} \end{array}\right\} \\
\mathfrak{F}^{\{Thm\}}_{\{Goal\},\{Prem\}} & = & \left\{Thm\text{: Conclusion matches } Goal \text{ with special algorithm}\right\} \\
 & & \left\{\begin{array}{ll} Acquisition\text{:} & \text{Conclusion contains } Closed \text{ as outermost} \\ & \text{predicate and a complex operation} \end{array}\right\} \\
\mathfrak{S}^{\{Prem\}}_{\{Goal,Thm\},\{\}} & = & \left\{Prem\text{: The nodes matching the premises of } Thm\right\}
\end{array}
$$

Figure 4.3: Argument agents for the *Closed* theorem cluster.

only those formulas as possible conclusions that contain an occurrence of the *Closed* predicate. We then have two function agents that try to match the theorems. The first tries to match the theorems *ClosedConst*, *ClosedFV*, and *ClosedSV* to the formulas accepted by the goal agent using first order matching. The second function agent uses the special algorithm instead of matching the theorems *ClComp*$\bar{+}$, *ClComp*$\bar{-}$, and *ClComp*$\bar{*}$ conventionally. Both function agents have an additional acquisition predicate specifying that the agents can acquire theorems whose conclusions have *Closed* as the outermost predicate and are in case of the first agent with respect to a simple operation and in case of the second agent with respect to a complex operation. The last agent is the generic support agent, which has an algorithm to extract the necessary premises from a matched theorem and, if there are any, tries to find appropriate proof lines containing them.

For our concrete example theorem the information that accumulates on the command blackboard for the *Closed* theorem cluster is as follows:



First the goal agent detects an occurrence of the *Closed* predicate in the given goal *Conc* and adds a PAI suggesting it as instantiation for *Goal* to the blackboard. With this PAI the function agents start matching their respective theorems to *Conc*, which is successful for the *ClComp*$\bar{+}$ theorem. The matched theorem is added as suggestion for the *Thm* slot of the PAI and the support agent starts its search. For the example the premises are $Closed(\mathbb{Z}_5, \lambda x_{\bullet}\lambda y_{\bullet}\bar{3}_5)$, $Closed(\mathbb{Z}_5, \lambda x_{\bullet}\lambda y_{\bullet}x\bar{*}y)$, and also to show the correctness of the sort assertion by proving $5 \in \mathbb{Z}$. Since we assumed that this is the first step in the proof the agent finds nothing and the second result PAI on the command blackboard is propagated to be suggested to the planner. In case the planner chooses to apply the *ClComp*$\bar{+}$ theorem in the current proof state the two premises of the theorem will become new open subgoals.

4.1.2.4 Discussion

The presented use of Ω-ANTS for suggesting assertion applications during proof planning enhances the traditional way of computing assertion applications in a method in several ways: It distributes the search for applicable assertions and decouples it from the planning algorithm. It also enables the use of specialized efficient algorithms for theorem matching. Furthermore, the mechanism does not have to have intrinsic information on the actual status of the knowledge base since the selection of theorems is done via the acquisition predicates of the agents rather than by reference to particular theorems. This also makes the approach dynamic in the sense that the mechanism can extend itself at runtime. However, one obvious disadvantage is that it requires the predicates for the goal and function agents to be explicitly specified and implemented, which requires some additional effort.

There are, of course, several alternative ways of enhancing the traditional assertion application we have considered.

The first is to incorporate Ω-Ants in a naïve way by automatically creating a goal agent for every available assertion, which can then match the assertion with the respective goal in each planning cycle. Results are then written on the single command blackboard for the *ApplyAss* method. An additional support agent could try to complete these results by looking for appropriate support lines that correspond to the premises of the suggested assertions. Once an applicable assertion has been found it can be signaled to the planner, which can then decide if and when to apply the *ApplyAss* method.

This approach has the advantage that all the goal agents can be automatically created and are all of the same form, which saves the effort of specifying and implementing both goal and function agents in our approach. We also need exactly one command agent, only. Furthermore, the search mechanism can be easily expanded since for every new theorem a new goal agent can be automatically added. In this scenario all goal agents use the same algorithm to match their assertion with the given goal. This could be a higher order matching algorithm since un-decidability is no longer a factor as agents that might commence an infinite computation are eventually stopped by Ω-Ants's resource mechanism. However, we could not incorporate specialized decidable algorithms for certain complex theorems. Another disadvantage of this approach is that in order to control the number of considered assertions they still have to be referred to explicitly — either via name or their theory — in the strategy by a control rule. Furthermore, to have one goal agent for each assertion has the effect that similar to the traditional method application, possibly all assertions are actually matched, even though in parallel and not sequentially. Although there might be criteria that can indicate that some theorems will not be applicable in the first place and could be excluded from the matching.

A second observation is that our presented use of Ω-Ants has certain resemblances to hashing [169] and term indexing techniques [103]. Here the approach is to have a hash-table that acts as a mapping from constants occurring in the knowledge base to the theorems in which those constants occur. This way the knowledge base can be quickly accessed by reference to constants occurring in the considered goal to narrow the number of theorems to be matched. And, since our reference objects are constant symbols and not the theorems themselves, the mechanism can be kept free from any inside-information on the status of the knowledge base. Although we have the appropriate higher order term indexing [127] available in Ωmega we chose not to implement this variant since one might want to impose stricter restrictions on the theorems that are considered apart from whether they have certain constant symbols in common with the current goal. Moreover, theorem matching has to be done again by a single algorithm, which is either not sufficiently powerful or undecidable. However, the function agents in our approach can naturally incorporate clever term indexing algorithms in their acquisition predicate to search the knowledge base.

Another possible approach is to enhance the original *ApplyAss* method by giving it a more powerful matching algorithm but not full higher order matching. Qualifying algorithms are, for instance, higher order pre-unification [175] or higher order pattern matching [174, 173]. The former is again undecidable and the latter has, albeit it is decidable, the limitations that theorems are often of a more recursive structure in order to be fit into pattern schemes.

4.2 Symbolic Computation in Proof Planning

In recent years there have been many attempts at combining computer algebra systems (CAS) and deduction systems (DS), either for the purpose of enhancing the computational power of the DS [107, 124, 17] or in order to strengthen the reasoning capabilities of a CAS [2, 18]. For the former integration there exist basically three approaches: (1) To fully trust the CAS, (2) to use the CAS as an oracle and to try to reconstruct the proof in the DS with purely logical inferences, and (3) to generate intermediate information output during a CAS calculation and to use this intermediary output to verify the computation. Following approach (1) one cannot guarantee the correctness of the proof in the DS any longer. While the correctness is no issue in approach (2) it foregoes the efficiency of a CAS and replaying the computation with purely logical reasoning might still impose a hard task on the DS. (3) is a compromise, where one can employ the computational strength of a CAS and additionally gain important hints to ease the reconstruction and checking of the computation.

We have, indeed, successfully experimented with idea (3) by implementing a prototype CAS ($\mu\mathcal{CAS}$) that consists of a small library of simple polynomial algorithms, which give us intermediate information on their computations [124, 193]. This intermediary information is used to derive abstract proof plans that can be transformed into proofs of the ΩMEGA system. Exploiting ΩMEGA's ability for step-by-step expansion of proof plans into natural deduction calculus proofs, the computations can be machine-checked in a fine-grained calculus level. While this way of integrating a computer algebra system into ΩMEGA solves the correctness issue, it has the drawback that there does not exist a full CAS that provides us with the necessary intermediary output on its calculations. As an alternative one could enrich the simplification mechanism of a regular CAS to output information on the applied rewriting rules. However, this is not only a non-trivial task itself but also falls short if analytical or numerical algorithms are involved in the simplification procedure, which are not based on rewriting rules.

In ΩMEGA we use a pragmatic approach to work around this problem in proof planning, which has originally been presented in [194]. It is based on the idea of HARRISON and THÉRY [107] that many hard symbolic computations are easy to check. We exploit this fact within the proof planning component of ΩMEGA: Results of non-trivial symbolic computations are used during the proof planning process. The verification of these calculations is postponed until a complete proof plan is refined to a low level calculus proof and it is arranged in a way, that we can use the trivial direction of the verification. This is achieved by using MAPLE [177] for computations during the planning process and $\mu\mathcal{CAS}$ to aid the verification.

Note that we do not use $\mu\mathcal{CAS}$ to verify the correctness of the algorithms involved but only of single instances of their computations. Thus, the produced proofs are proofs for the computation of existential witnesses, only, and not for the overall algebraic procedure. Note also that we do not only verify algebraic solutions of algorithms but also analytic or even numerical solutions.

4.2.1 Integration of Computer Algebra

In this section we first present the general architecture for the integration of computer algebra into ΩMEGA. For a more detailed introduction see also [124, 193].

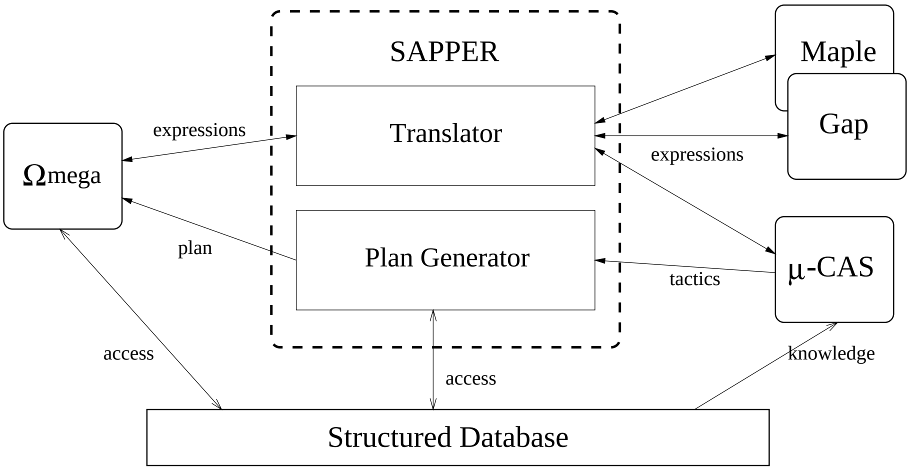


Figure 4.4: Interface between Ωmega and computer algebra systems

Then we elaborate our new approach for integrating symbolic computations and their verification into proof planning in Ωmega.

4.2.1.1 Architecture

The integration of computer algebra into Ωmega is accomplished by the sapper system [193], which can be seen as a generic interface for connecting one or several computer algebra systems (see figure 4.4). An incorporated CAS, like Maple [177], Gap [94], or $\mu\mathcal{CAS}$ [193], is treated as a slave to Ωmega. This means only Ωmega can call the CAS but not vice versa. From the technical point of view, Ωmega and the CASs are independent processes while the interface is a process providing a link for communication. Its role is to automate the broadcasting of messages by transforming output of one system into data that can be processed by the other.[2] The maintenance of processes and passing of messages is managed by the MathWeb [88] environment into which Ωmega is embedded.

The role of sapper in the integration has two distinct aspects: Firstly, arbitrary CASs can be easily used as black box systems for term rewriting (similar to the approaches of [18, 17]) and sapper works as a simple bridge between the planner and the CASs. Secondly, sapper also offers means to use a CAS as a proof planner; that is, if the CAS can provide additional information on its computations, this information is recorded by sapper and translated into a sequence of tactics that can eventually verify the computation. Since there does not exist a state-of-the-art system that provides this information, we use our own $\mu\mathcal{CAS}$ system, that is a collection of simple algorithms for arithmetic simplification and polynomial manipulations including a plan generating mode (see [124]).

The two tasks of a CAS, rewriting and plan generation, are mirrored in the interface (see figure 4.4) that basically can be divided into two major parts; the *translator* and the *plan generator*. The former performs syntax translations between Ωmega and a CAS in both directions while the latter only transforms intermediate output of $\mu\mathcal{CAS}$ to Ωmega proof plans. Figure 4.4 also depicts the different uses of the two CAS involved: While Maple and Gap are connected as black box systems, only, $\mu\mathcal{CAS}$ can be used both as black box and as plan generator. Although

[2] This is an adaptation of the general approach on combining systems in [64].

figure 4.4 only shows three computer algebra systems, the interface is not restricted to them and can also connect to any other CAS. In fact, MAGMA is also currently interfaced with ΩMEGA.

While the translation part of the interface is commonplace, the plan generator is the actual specialty of SAPPER. It provides the machinery for the proof plan extraction from the specialized algorithms in $\mu\mathcal{CAS}$. These are equipped with a *proof plan generating mode* that returns information on single steps of the computation within the algorithms. The output produced by the execution of a particular algorithm is recorded by the plan generator, which converts it, according to additional information in the proof, into a proof plan. In order to produce meaningful information $\mu\mathcal{CAS}$ needs to have a certain knowledge about the proof methods and tactics available to ΩMEGA in its knowledge base. Thus, references to logical objects (methods, tactics, theorems, or definitions) of the knowledge base are compiled a priori into the algebraic algorithms for documenting their calculations. SAPPER's plan generator uses produced intermediary output to look up tactics and theorems of an ΩMEGA theory (see figure 4.4) in order to assemble a valid proof plan.

To implement a plan generating mode is a simple task for simple CAS algorithms. An algorithm has to be enriched to produce output that indicates the computations performed in crucial points. This output then has to refer to tactics in ΩMEGA's knowledge base that correspond to the computational steps. Thus, to extend an algorithm with a plan generating mode generally also involves writing appropriate tactics in ΩMEGA.

4.2.1.2 Integration into Proof Planning

In proof planning we can use symbolic calculations in two ways: (1) In control rules hints are computed to help guiding the planning process, and (2) within method-applications complicated algebraic computations can be carried out by computer algebra systems to simplify the proof. As a side-effect both cases can restrict possible instantiations of meta-variables.

An example for case (1) are control rules that suggest meta-variable constraints by computing possible instantiations using a computer algebra system. However, the computed instance is regarded only as a hint; that is, in case the planning attempt fails with this particular instantiation the planner can still backtrack and proceed by using regular search. This way the verification of the hint is done by the subsequent proof planning. We will see examples for this in chapters 7 and 8 where, for instance, GAP is employed to suggest instantiations of meta-variables in the context of proofs in the residue class domain.

Case (2) is a way to simplify proofs by incorporating a symbolic computation directly as a single step in the proof. During the application of a method a computer algebra system is called and its results are directly incorporated into the proof plan or a proof line is justified by the fact that the computation succeeds. Here the computation is no longer treated as a hint but rather assumed to be correct for the time being. Thus, the constructed proof plan is only correct provided the computation is correct. But we have to keep in mind that in ΩMEGA all plans have to be expandable to ND calculus proofs. However, using a system whose computations are checkable, like $\mu\mathcal{CAS}$, restricts us to the use of its rather simple algorithms, which might not always be sufficient for the task at hand. What we really would like, is to combine the computational power of a CAS like MAPLE to perform non-trivial computations with the verification strength of $\mu\mathcal{CAS}$.

Therefore, we try to exploit as much as possible the fact that many difficult symbolic computations are easy to verify. This is folklore in mathematics and has already been elaborated in [107]; the most prominent example for this is certainly indefinite symbolic integration, which is still a hard task for many CASs. Results of indefinite symbolic integration algorithms are, however, easily checked, since it involves only differentiation of the result and comparison with the original function. The comparison might be less trivial if the system involved does not have canonical representations of its terms. Then the equivalence of the integrated function and the result of the differentiation has to be shown separately. Other examples are the computation of roots of functions or factorization of polynomials, which involve non-trivial algorithms, but the verification of the results only involves straight-forward arithmetic.

The separation of computation and verification can be easily achieved within proof planning: During the planning process the applicability of a method is solely determined by matching and checking the application conditions. As mentioned earlier, the latter can be used to execute arbitrary functions, therefore we can also implement conditions that call MAPLE and in case useful results are returned, bind these to some method parameters. During the planning process we are not concerned with the verification of the computation, and postpone it until the method is actually expanded. This is done by stating a rewriting step that is justified by the application of a CAS within the proof schema of the method, preferably in those lines that are introduced during the expansion of the method.

Thus, we design our planning methods in a way that MAPLE is called in one of the application conditions to perform the difficult computations during the planning process. The proof schema then contains the appropriate proof steps that enable the application of $\mu\mathcal{CAS}$ to verify MAPLE's computation during the refinement of a proof plan, in the easier direction.

4.2.1.3 Dealing with Different Canonical Forms

When using a system such as MAPLE for a computation within some method and $\mu\mathcal{CAS}$ to verify MAPLE's result we might have problems identifying the term resulting from $\mu\mathcal{CAS}$'s computation with the original term MAPLE was applied to. Thus, we have to take care of the problem of distinct canonical forms of the systems involved during the expansion of a computation. Note that also MAPLE works with canonical representations of terms[3] for the following investigations it is only important that $\mu\mathcal{CAS}$ has unique canonical forms.

Let Φ_0 be the original term in the proof, while Φ_{MAPLE} denotes the term that results from applying MAPLE to Φ_0, and let $\Phi_{\mu\mathcal{CAS}}$ be the term returned by $\mu\mathcal{CAS}$ applied to Φ_{MAPLE}. Furthermore, let $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$ be the sequence of tactics computed by $\mu\mathcal{CAS}$ whose application to Φ_{MAPLE} yields the proof plan (4.1).[4]

$$\Phi_{\text{MAPLE}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \ldots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}}. \tag{4.1}$$

We then have three cases to consider:

(a) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ coincide,

(b) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ are distinct, however Φ_0 occurs at some point during the expansion, and

[3] The form of MAPLE's result may vary for equivalent arithmetic expressions (in two different runs of MAPLE), depending on the form of the input. For instance, MAPLE's simplification of $x + 2z + y - z$ yields $x + z + y$, while the same computation with input $x + y + 2z - z$ would yield $x + y + z$ in a different run. See also [1] on this point.

[4] For the sake of clarity, we omit any context the terms Φ_j might be embedded in; that is, we view the proof plan as rewriting steps of a sub-term of some arbitrary formula.

(c) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ are distinct, and Φ_0 does not occur during the expansion.

Case (a) is trivial. Case (b) means that we have some $1 \leq i \leq n$, such that

$$\Phi_{\text{Maple}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \ldots \xrightarrow{\mathcal{T}_i} \Phi_0 \xrightarrow{\mathcal{T}_{i+1}} \ldots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}}. \tag{4.2}$$

This problem can be easily solved by successively applying the single tactics and checking after each application whether the resulting term is already equivalent to Φ_0. In this case the proof can be concluded directly. The remainder of the tactic sequence, $(\mathcal{T}_{i+1}, \ldots, \mathcal{T}_n)$ in (4.2), is discarded.

Case (c) is less trivial since the produced tactics are not sufficient to fully justify the computation and thus we are left with a new proof problem, namely to derive the equality of $\Phi_{\mu\mathcal{CAS}}$ and Φ_0. However, at this point we can make use of the lexicographic term ordering of $\mu\mathcal{CAS}$: If $\Phi_{\mu\mathcal{CAS}}$ and Φ_0 really constitute the same arithmetic expression, applying $\mu\mathcal{CAS}$ simplification algorithm to Φ_0 will yield $\Phi_{\mu\mathcal{CAS}}$. Note that this step might not only include trivial reordering of a sum but can contain more sophisticated arithmetic. The execution of the simplification algorithm will then return a sequence of tactics $(\mathcal{S}_1, \ldots, \mathcal{S}_m)$ that results in:

$$\Phi_{\text{Maple}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \ldots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}} \xleftarrow{\mathcal{S}_m} \ldots \xleftarrow{\mathcal{S}_1} \Phi_0 \tag{4.3}$$

In practice, we deal with this problem slightly different, since in ΩMEGA's tactic expansion mechanism calls to $\mu\mathcal{CAS}$ have to be carried out explicitly by expanding the according justification, and not implicitly during an expansion itself. Thus, we introduce a new subproof for the equality of $\Phi_{\mu\mathcal{CAS}}$ and Φ_0:

$$\begin{array}{ll} \Phi_{\mu\mathcal{CAS}} = \Phi_{\mu\mathcal{CAS}} & \quad (=\mathit{Ref}) \\ \Phi_{\mu\mathcal{CAS}} = \Phi_0 & \quad (\mathit{CAS}) \end{array}$$

The first line is an instance of reflexivity of equality, an axiom of ΩMEGA's basic calculus. The equation of the second line serves then to apply a rule of equality substitution (=*Subst*) to finish the original expansion, resulting in proof plan (4.4).

$$\Phi_{\text{Maple}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \ldots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}} \xrightarrow{=Subst} \Phi_0. \tag{4.4}$$

In order to completely verify the computation the justification (CAS) above must be expanded as well. This results in the second call to $\mu\mathcal{CAS}$, yielding a proof plan equivalent to the right hand side of (4.3).

4.2.2 Example

We illustrate our approach with an example of a general proof planning method for solving equations. This method is also used in the proofs of the case studies we shall present in part III of this thesis.

Figure 4.5 depicts the planning method *SolveEqu* whose purpose is to solve an equational subgoal if possible. The method has only one conclusion, which will be removed from the planning state if the application of the method is successful.

Thus, *SolveEqu* is handled by the planner as follows: If an open line in the planning state contains an equation and therefore matches L_1, *SolveEqu*'s parameters Φ and Ψ are instantiated. Then the application condition is evaluated. `Solve-with-Maple` calls MAPLE to compute a solution of the equation $\Phi = \Psi$ using MAPLE's function `solve`. The terms Φ and Ψ are translated into the appropriate MAPLE syntax: Arithmetic functions in ΩMEGA are translated into the corresponding arithmetic functions in MAPLE and digits are mapped to digits and all other

Method: *SolveEqu*	
Premises	
Appl. Cond.	`Solve-with-Maple`$(\Phi = \Psi)$
Conclusions	$\ominus L_1$
Declarative Content	(L_0) $\Delta \vdash \Phi = \Phi$ $(=\!Ref\ \Phi)$ (L_1) $\Delta \vdash \Phi = \Psi$ $(CAS\ L_0\ \langle 2 \rangle)$

Figure 4.5: The *SolveEqu* method.

occurring terms are translated into Maple variables. In particular non-arithmetic functions in the ΩMEGA expression become single variables in Maple representation.

In the case where Maple returns a general solution for the equation the method is applicable. Here, general solution means that all variables contained in the expression can be arbitrarily instantiated and, in particular, do not depend on each other. However, if the original ΩMEGA equation contains meta-variables we allow for specific solutions to the equation as long as only the Maple variables corresponding to the meta-variables have specific or dependent solutions, while the solution is general for all other variables. The specific solutions for the meta-variables are then used to instantiate or further constrain the meta-variables in question. In the case where Maple does not return a general solution in the above sense or does not return any solution at all, the application of *SolveEqu* fails.

If the application condition is successfully evaluated, the instantiated method is introduced into the partial proof plan and the goal L_1 is removed as a planning goal. Since *SolveEqu* does not introduce any new subgoals the particular subproblem the goal L_1 constituted is fully justified. When an application of the method is expanded later on, the subproof given in the declarative content is introduced; that is, the sequent L_0 is newly added to the proof. This line serves to certify the correctness of the solution computed by Maple by making this computation explicit. Here the equality of Φ and Ψ is derived from the reflexivity of equality. The newly introduced justification of line L_1 indicates both that the step has been introduced by the application of a CAS and that its expansion can be realized by using $\mu\mathcal{CAS}$ in plan generating mode. Furthermore the given term position 2 indicates that for the verification of Maple's computation it is necessary to certify that Ψ can successfully be transformed into Φ.

To perform the verification, we must use basic arithmetic, only, instead of the generally harder problem of solving an equation in arbitrarily many variables that was performed by Maple. Thus, we can use $\mu\mathcal{CAS}$'s simplification component for the verification. For a concrete instance, $\mu\mathcal{CAS}$ would return a sequence of tactics indicating single computational steps that have been performed inside the computer algebra algorithm. This proof plan is then inserted into the proof and further expanded to show the correctness of the computation.

But the verification of the performed computation can fail since the application of the *SolveEqu* method can be faulty. The method itself can be used in various domains, for instance, in the following example it is applied to an equation of integer. The computation in Maple, however, is not restricted to a particular domain. Therefore, Maple tries to solve the equation over the complex numbers. Consequently, a success of Maple does not necessarily entail that the equation in question actually has a solution in the considered domain. The expansion of the method can thus fail for two reasons: Either the proof plan returned by $\mu\mathcal{CAS}$, whose algorithms also work over the complex numbers, contains a tactic that is not known in the actual problem domain. Or during the expansion of $\mu\mathcal{CAS}$'s proof plan a tactic or a theorem cannot be applied with terms computed by MAPLE. For instance, MAPLE can return a meta-variable instantiation containing a rational number, which cannot be instantiated in a theorem ranging over the integers.

As a concrete example we consider the application of *SolveEqu* in the proof of associativity of the operation $\lambda x_\nu . \lambda y_\nu . (x * y) * 2$, where $*$ is the multiplication on integers. This is part of problems we shall examine in the case study in chapter 7 in the context of residue classes. However, at this point we are not concerned with the actual problem and only concentrate on the part of the proof that we need to demonstrate the verification technique we present. Moreover, for the sake of the example, the problem is slightly changed by introducing a meta-variable. Thus, the task at hand is the solution of the equation line

$$L_1. \vdash (a * ((b * c) * 2)) * 2 = (((a * mv) * 2) * c) * 2 \quad (Open)$$

Here, mv is a meta-variable and a, b, c are arbitrary constants. When *SolveEqu* is applied the function call passed to MAPLE is:

```
solve((a*((b*c)*2))*2=(((a*mv)*2)*c)*2);
```

In case any of the constants would have been a non-arithmetical function, say $f(x)$, then it would have been transformed to f_x and treated as a MAPLE variable. For the equation MAPLE returns three possible solutions among which one is of the form $\{\mathtt{a} = \mathtt{a}, \mathtt{c} = \mathtt{c}, \mathtt{b} = \mathtt{b}, \mathtt{mv} = \mathtt{b}\}$. This corresponds to a general solution in the variables a, b and c and a specific solution for the meta-variable mv. Hence the method is applicable which results in the proof line

$$L_1. \vdash (a * ((b * c) * 2)) * 2 = (((a * b) * 2) * c) * 2 \quad (SolveEqu)$$

The application of *SolveEqu* binds the meta-variable mv to b. Here we have a point to possibly introduce an error into the proof in case mv is bound to a term representing a non-integer value. However, since b is a general variable the following expansion of the method goes through smoothly.

Upon expansion of the justification the declarative content of the *SolveEqu* method is introduced giving us the two lines

$$\begin{array}{ll} L_2. \vdash (a * ((b * c) * 2)) * 2 = (a * ((b * c) * 2)) * 2 & (=Ref) \\ L_1. \vdash (a * ((b * c) * 2)) * 2 = (((a * b) * 2) * c) * 2 & (CAS\ L_2\ \langle 2 \rangle) \end{array}$$

Here the second line is justified by the application of a CAS. In order to obtain a pure ND-level proof this line needs to be further expanded. However, since during the application of *SolveEqu* the equation was solved by MAPLE, we do not have any additional information for an expansion. To justify the computation in more detail we use an algorithm within our $\mu\mathcal{CAS}$ system in plan generation mode that produces a trace output that gives more detailed information on single computational steps. Instead of simulating the algorithm for solving the equation as a whole within $\mu\mathcal{CAS}$, we simply use an algorithm that simplifies the term on the right-hand side of the equation. Thus, $\mu\mathcal{CAS}$ verifies the result of MAPLE's computation with the help of a simpler algorithm. The yielded proof plan consists of a sequence of tactics indicating single computational steps of the algorithm. Within the $\mathcal{PDS}$, the single step can be expanded to a plan with higher granularity. The newly introduced proof steps are:

$$\begin{array}{lll}
L_6. & \vdash \quad 4*(a*(b*c)) = 4*(a*(b*c)) & (=\!\mathit{Ref}) \\
L_5. & \vdash \quad 4*(a*(b*c)) = (a*((b*c)*2))*2 & (\mathit{CAS}\ \langle 2\rangle) \\
L_2. & \vdash (a*((b*c)*2))*2 = (a*((b*c)*2))*2 & (=\!\mathit{Ref}) \\
L_4. & \vdash (a*((b*c)*2))*2 = 4*(a*(b*c)) & (=\!\mathit{Subst}\ L_2\ L_5\ \langle 2\rangle) \\
L_3. & \vdash (a*((b*c)*2))*2 = (2*(a*(b*c)))*2 & (\mathit{PullDigit}\ L_4\ \langle 2\rangle) \\
L_1. & \vdash (a*((b*c)*2))*2 = (((a*b)*2)*c)*2 & (\mathit{PullDigit}\ L_3\ \langle 2.1\rangle)
\end{array}$$

The lower three lines correspond to the step-by-step computations of the $\mu\mathcal{CAS}$ algorithm, which descends recursively into the term, pulls all numbers to the front of the term to compute the coefficient and sorts all remaining sub-terms lexicographically. In our example the sub-terms are already in a lexicographic order, however, the algorithm normalizes the coefficient in two steps as indicated by the two applications of the *PullDigit* tactic. First, *PullDigit* is applied at position $\langle 2.1\rangle$ of the equation which rewrites the sub-term $(((a*b)*2)*c)$ to $(2*(a*(b*c)))$ on the righthand side of the equation. The second application of *PullDigit* sorts the second occurrence of 2 to the front and multiplies it to compute the actual coefficient 4.

Since $\mu\mathcal{CAS}$'s simplification algorithm yields only $4*(a*((b*c)))$ as a result we have a conflict of canonical forms, as described in the preceding section. Therefore, the upper two lines have to be introduced in the proof in order to justifiy the equality substitution ($=$*Subst*). The new *CAS* justification can be expanded with another call to $\mu\mathcal{CAS}$. However, we want to focus on the expansion of the original proof plan contained in lines L_1, L_3, L_4. So far the expansion of the original *CAS* justification has been exclusively done by $\mu\mathcal{CAS}$ proof plan generation mode. At this stage $\mu\mathcal{CAS}$ cannot provide any more details about the computation and the subsequent expansion of the next hierarchic level can be achieved without further use of a CAS. Let us for instance take a look at the expansion of the first application of the *PullDigit* tactic, which basically describes the reordering within a product:

$$\begin{array}{lll}
L_3. & \vdash (a*((b*c)*2))*2 = (2*(a*(b*c)))*2 & (\mathit{PullDigit}\ L_4\ \langle 2\rangle) \\
L_8. & \vdash (a*((b*c)*2))*2 = (2*((a*b)*c))*2 & (\mathit{Assoc}{*}\ L_3\ \langle 2.1.2\rangle) \\
L_7. & \vdash (a*((b*c)*2))*2 = ((2*(a*b))*c)*2 & (\mathit{Assoc}{*}\ L_8\ \langle 2.1\rangle) \\
L_1. & \vdash (a*((b*c)*2))*2 = (((a*b)*2)*c)*2 & (\mathit{Commu}{*}\ L_7\ \langle 2.1.1\rangle)
\end{array}$$

Here the tactics named *Assoc∗* and *Commu∗* correspond to the application of the theorems of associativity and commutativity of times as a rewrite rule. Now the subproof introduced when expanding *PullDigit* is already on the level of applications of basic laws of arithmetic. These tactics can, however, be expanded even further. Expanding, for example, the *Commu∗* justification yields:

$$\begin{array}{lll}
L_9. & \vdash \quad \forall x{:}\mathbb{Z}.\forall y{:}\mathbb{Z}.x*y = y*x & (\mathit{Theorem}) \\
L_{10}. & \vdash \quad \forall y{:}\mathbb{Z}.(a*b)*y = y*(a*b) & (\forall_E\ L_9\ (a*b)) \\
L_{11}. & \vdash \quad (a*b)*2 = 2*(a*b) & (\forall_E\ L_{10}\ 2) \\
L_7. & \vdash (a*((b*c)*2))*2 = ((2*(a*b))*c)*2 & (\mathit{Assoc}{*}\ L_8\ \langle 2.1\rangle) \\
L_1. & \vdash (a*((b*c)*2))*2 = (((a*b)*2)*c)*2 & (=\!\mathit{Subst}\,L_7\,L_{11}\ \langle 2.1.1\rangle)
\end{array}$$

This last expansion step details the application of commutativity of addition as rewrite step by deriving the right instance from the theorem of commutativity.

At this point we have already expanded to very fine-grained level of the proof. But we have seen in chapter 2 that equality is a defined concept in ΩMEGA. Therefore, tactics such as $=$*Subst* and $=$*Ref* can also be expanded in order to justify equational reasoning by Leibniz-equality and to be able to fully proof check the computation. However we omit the tedious details of these expansions here. Provided we have carried out those expansions as well, the proof checker approves, and we have correct proofs for all the applied theorems in our database, we have successfully verified the particular computation which guarantees the correctness of the overall proof.

4.2.3 Discussion

From our current experience (see [194] and part III), the presented approach is well suited for symbolic computations whose verification is relatively trivial, for instance, where only simple arithmetic needs to be employed. However, the method is not feasible for computations where the verification is as expensive or even more complicated than the computation itself. At least in the latter case it might be more practicable to immediately specify the computation as a $\mu\mathcal{CAS}$ algorithm. Computations where the verification will be definitely non-trivial are those involving certain uniqueness properties of the result. For instance, when employing MAPLE to compute all roots of a function, it will be a hard task to verify that there exist no more roots than those actually computed. For further discussion of this point we refer to [107].

Although we presented our ideas in this paper in the context of proof planning, we strongly believe that the approach could also work in tactical (interactive) theorem proving. One necessary prerequisite will be the existence of an explicit proof object for storing proof steps that contain calculations. These steps can then be verified with the help of the simple $\mu\mathcal{CAS}$ algorithm. Even if the proof object does not have the advanced facilities for step-wise expansion of proof steps the verification could be done by transforming $\mu\mathcal{CAS}$ output into tactics, and thereby primitive inferences, of the respective system. Those primitive inferences would not necessarily have to be incorporated into the proof object. For systems not maintaining explicit proof objects, such as HOL [102] or PVS [165], the approach of [107] would suit best. Here the symbolic computations are verified immediately by tactics build on primitive inferences of HOL. However, this approach directly implements the verification algorithms as tactics in the HOL system as correspondences to MAPLE's computation.

4.3 Summary of Chapter 4

This chapter was concerned with the integration of different reasoning techniques within a proof planning framework. In particular we presented how the Ω-ANTS mechanism can be fruitfully employed in MULTI for interactive proof planning and for efficiently determining the applicability of assertions in a proof. For the former Ω-ANTS can be used as an algorithm within MULTI and then parameterized with a regular strategy. Thereby we generally implement the interactive strategy using Ω-ANTS as a complement to the corresponding automatic strategy using the regular planning algorithm *PPlanner*. This enables to switch freely between interactive and automatic proof planning.

A second application of Ω-ANTS in MULTI is to determine and suggest possible assertion applications. Here the mechanism is used to automatically create meaningful clusters of theorems available in ΩMEGA's knowledge base. Applicability of assertions is then determined constantly in the background and signalled to the planner indepedent from regular method matching. Theorems can also be dynamically added and are automatically integrated into the mechanism. This use of

Ω-Ants has the advantage that the search for applicable assertions can be decoupled from the actual proof planning process. Moreover, the available theorems are organized in clusters which enables to filter theorem clusters with respect to their relevance for a given goal and thus to avoid blind search for applicable theorems. The clusters are automatically formed with respect to given criteria independent from the actual status of the knowledge base and can be dynamically extended.

We have also seen how computer algebra systems are integrated into Ωmega in order to prune the search space during proof planning. Thereby symbolic computations can be used in two ways: In control rules to compute hints for meta-variable instantiations and in methods to perform rewriting and thereby shorten the proofs. While the correctness of the hints is automatically checked by the proof planner, the correctness of computations inside methods has to be explicitly verified during the refinement of a proof plan.

Thereby we make use of the following technique: During the planning process we employ a regular full-grown CAS that allows us to perform non-trivial computations. When refining a constructed proof plan to an actual calculus level proof the rewrite step introduced by the CAS has to be expanded into low level logic derivations. This is done with the help of a small self-tailored CAS called $\mu\mathcal{CAS}$ that provides detailed information on its computations in order to construct the expansion of the rewrite step. Since $\mu\mathcal{CAS}$ is specialized currently only on arithmetic we can so far incorporate only those computations whose verification involves arithmetic.

Part III

Case Studies

Chapter 5

Equivalence and Uniqueness Proofs

In this chapter we demonstrate the use of Ω-Ants to automatically construct proofs as sketched in chapter 3.5. We shall model an efficient and goal directed search procedure for the first order fragment of our ND calculus in Ω-Ants. This fragment is enriched by inference rules that incorporate automated theorem provers and that deal with definitions and the description operator. Using the automation wrapper proofs can then be automatically constructed. The time bound of the automation wrapper is particularly influential as to whether and at what point a prover can find a proof and variations of the time bound can change the shape of the constructed proof. The examples we shall consider are proofs of equivalence of different definitions of a group, uniqueness proofs and some simple theorems from group theory.

The chapter is organized as follows: We first give some definitions of algebraic structures and in particular several alternative definitions of a group. We shall then formulate the example theorems before painstakingly formalizing all necessary concepts. In order to prove the theorems we introduce an efficient and goal directed search procedure for natural deduction calculus we have modeled in Ω-Ants and enrich it by some special rules to deal with both definitions and description and to apply external automated theorem provers. Then we demonstrate the working scheme of the proofs for the given theorems with a simple example. We conclude the chapter by giving a challenging example theorem about the equivalence of two alternative definitions of a group, for which, to the knowledge of the author, current state of the art automated theorem proving techniques fail.

5.1 Some Definitions

In this section we introduce some definitions of algebraic structures. In particular we give several equivalent definition of the notion of a group. We start by giving a classical group definition.

Definition 5.1 (Group): Let G be a nonempty set and let $\cdot$ be a binary mapping on G. G is a *group* if the following holds:

$\mathcal{G}1$) For all $a, b \in G$ holds $a \cdot b \in G$. (Closure)

$\mathcal{G}2$) For all $a, b, c \in G$ holds $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. (Associativity)

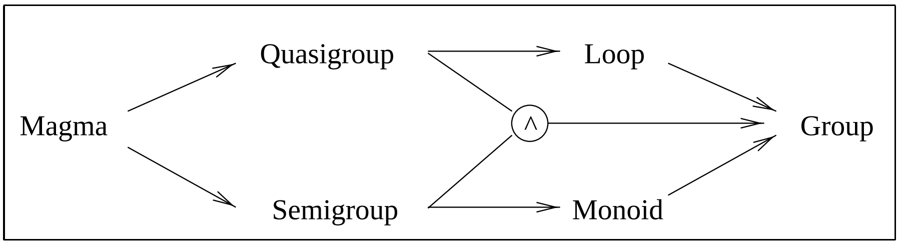


Figure 5.1: Constructing groups via more general algebras

$\mathcal{G}3$) There exists $e \in G$ such that for all $a \in G$ holds $e \cdot a = a \cdot e = a$. (Existence of a unit element)

$\mathcal{G}4$) For all $a \in G$ exists $x \in G$ such that $a \cdot x = x \cdot a = e$. (Existence of inverses)

□

We call the element $e \in G$ in axiom $\mathcal{G}3$ the *unit element* or the *identity* of the group. An element $x \in G$ that satisfies property $\mathcal{G}4$ is called the inverse of $a \in G$.

The definition given in 5.1, however, is not minimal, since we can define a group also if we replace properties $\mathcal{G}3$ and $\mathcal{G}4$ with their weaker, more general forms that postulate the existence of only a left identity and left inverse elements:

$\mathcal{G}3^*$) There exists $e \in G$ such that for all $a \in G$ holds $e \cdot a = a$. (Existence of a left unit element)

$\mathcal{G}4^*$) For all $a \in G$ exists $x \in G$ such that $x \cdot a = e$. (Existence of left inverses)

Naturally, the choice of using the left identity and left inverse is arbitrary since we can analogously define a group using right identity and right inverses or even alternate the sides of identity and inverses.

We can also substitute both $\mathcal{G}3$ and $\mathcal{G}4$ with a single property we refer to as the existence of divisors and gain an even shorter definition of the notion of a group.

$\mathcal{G}3'$) For all $a, b \in G$ exist $x, y \in G$ such that $a \cdot x = b$ and $y \cdot a = b$. (Existence of Divisors)

One can show the equivalence of the different kind of definitions without much effort. Formalizing these proofs or even automating them is, however, far less trivial.

We can also define the notion of a group via larger, less concrete algebraic structures. Hereby we have several possible ways to arrive at a definition as is outlined in figure 5.1. We shall first give the definitions of the different algebraic structures involved, especially since they are also relevant for the case study we present in chapter 7, before we give several equivalent definitions of a group based on these structures.

Starting on the left side of the outline given in figure 5.1 we start by defining the most general of our algebraic structures, a magma. Magmas are also sometimes called groupoids or multiplicative sets.

Definition 5.2 (Magma): Let M be a nonempty set together with a mapping $\cdot : M \times M \to M$ that uniquely appoints to every two elements in M a third element in M. The structure $(M, \cdot)$ is called a *magma*. □

A magma corresponds thus to an algebra having the closure property $\mathcal{G}1$ from definition 5.1. We first follow the lower branch in our hierarchy of algebraic structures of figure 5.1.

Definition 5.3 (Semi-Group): Let $(S, \cdot)$ be a magma. S is a *semi-group* if for all $a, b, c \in S$ holds $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. □

The properties of a semi-group are equivalent to the group properties $\mathcal{G}1$ and $\mathcal{G}2$. If we add also property $\mathcal{G}3$ we arrive at the notion of a monoid.

Definition 5.4 (Monoid): A semi-group $(M, \cdot)$ is a *monoid* if there exists an element $e \in M$ such that for all $a \in M$ holds $a \cdot e = e \cdot a = a$. □

Naturally, if we have a monoid with the final group property $\mathcal{G}4$ we arrive at the definition of a group. This concludes the lower branch in figure 5.1.

If we now start anew with a magma but follow the upper branch we first define the notion of a quasi-group.

Definition 5.5 (Quasi-Group): A magma $(Q, \cdot)$ is called a *quasi-group* if for all $a, b \in Q$ exist $x, y \in Q$ such that $a \cdot x = b$ and $y \cdot a = b$ hold. □

A quasi-group is thus an algebraic structure with the two group properties $\mathcal{G}1$ and $\mathcal{G}3'$. The multiplication table of a finite quasi-group is also called a *Latin square* and has the property that in each row and each column each element of the quasi-group occurs exactly once. If we now require a quasi-group also to have a unit element (i.e., property $\mathcal{G}3$) we arrive at the definition of a loop.

Definition 5.6 (Loop): Let $(L, \cdot)$ be a quasi-group. L is a *loop* if there exists an element $e \in L$ such that for all $a \in L$ holds $a \cdot e = e \cdot a = a$. □

To conclude the upper branch we can define a group as an associative loop. Apart from following the lower or the upper branch of our outline we can also take the middle course by defining a group as being both a semi-group and a quasi-group. This corresponds to the group properties $\mathcal{G}1$, $\mathcal{G}2$, and $\mathcal{G}3'$.

We summarize the different possible ways to define a group in the following assertion:

Assertion 5.7: *Group Let G be a nonempty set and let $\cdot$ be a binary operation on G. The following assertions are equivalent:*

(i) *$(G, \cdot)$ is a group.*

(ii) *$(G, \cdot)$ is a monoid and every element of $a \in G$ has an inverse.*

(iii) *$(G, \cdot)$ is a loop and $\cdot$ is associative.*

(iv) *$(G, \cdot)$ is both a quasi-group and a semi-group.*

5.2 Some Theorems

In this section we shall present two classes of theorems. The first are essentially the equivalent statements from the preceding section. The second are several uniqueness theorems involving some of the algebraic structures introduced above and additionally two rather simple theorems from group theory.

1. $[\exists \circ \textbf{.} Group(G,\circ)] \Leftrightarrow [\exists \star \textbf{.} NonEmpty(G) \wedge Closed(G,\star) \wedge Assoc(G,\star)$
$\wedge[\exists e{:}_G \textbf{.} LeftUnit(G,\star,e)]$
$\wedge LeftInverse(G,\star,LeftStructUnit(G,\star))]$

2. $[\exists \circ \textbf{.} Group(G,\circ)] \Leftrightarrow [\exists \star \textbf{.} NonEmpty(G) \wedge Closed(G,\star)$
$\wedge Assoc(G,\star) \wedge Divisors(G,op)]$

3. $[\exists \circ \textbf{.} Group(G,\circ)] \Leftrightarrow [\exists \star \textbf{.} Monoid(G,\star) \wedge Inverse(G,\star,StructUnit(G,\star))]$

4. $[\exists \circ \textbf{.} Group(G,\circ)] \Leftrightarrow [\exists \star \textbf{.} Loop(G,\star) \wedge Assoc(G,\star)]$

5. $[\exists \circ \textbf{.} Group(G,\circ)] \Leftrightarrow [\exists \star \textbf{.} Quasigroup(G,\star) \wedge Semigroup(G,\star)]$

Table 5.1: Some theorems on the equivalence of group definitions.

5.2.1 Equivalence Theorems

The first set of theorems we consider is given in table 5.1 and is concerned with equivalences of different definitions of a group. The formal concept $Group(G,\circ)$ is our reference definition of a group, which is similar to the one given in definition 5.1 in the preceding section. Note that the theorems are concerned with the equivalence of two different structures given in the form of the same set G and two different operations given as separately quantified variables $\circ$ and $\star$.

The first theorem states the equivalence between the reference definition and a definition postulating the existence of a left unit element and left inverses. Thereby the expression

$$LeftInverse(G,\star,LeftStructUnit(G,\star))$$

means that for each element of G there exists an inverse with respect to the left unit element of the structure $(G,\star)$. The term $LeftStructUnit(G,\star)$ references thus to the actual unit element. This element has to be uniquely determinable since the definition $LeftStructUnit$ uses the description operator as we shall see later on. Formulating it this way, enables us to refer to the unit element in G from arbitrary sub-formulas without having to state it explicitly. In the case of theorem 1 we could have also directly used the explicitly given unit element by stating the left inverse property in the scope of the existential quantification:

$$\exists e{:}_G \textbf{.} [LeftUnit(G,\star,e) \wedge LeftInverse(G,\star,e)]$$

However, this is not always possible as we can easily observe in theorem 3, which states claim (ii) from assertion 5.7. Here we have to refer to the unit element of the monoid $(G,\star)$, which can only be done using the reference term $StructUnit(G,\star)$, since the actual requirement of the unit element is buried inside the abstract concept $Monoid$.

The remaining theorems of table 5.1 state that the standard definition is equivalent to the definition consisting of group properties $\mathcal{G}1$, $\mathcal{G}2$, and $\mathcal{G}3'$ (theorem 2) and theorems 3 to 5 correspond to the single equivalences claimed in assertion 5.7.

5.2.2 Uniqueness and Other Theorems

Table 5.2 depicts six theorems, where the first four state uniqueness properties and the latter two are simple statements in group theory. In detail theorems 1 to 3

1. $\mathit{Quasigroup}(Q,\circ) \Rightarrow \forall a{:}{\scriptstyle Q}.\forall b{:}{\scriptstyle Q}.[\exists! x{:}{\scriptstyle Q}.ax = b] \wedge [\exists! y{:}{\scriptstyle Q}.ya = b]$
2. $\mathit{Monoid}(M,\circ) \Rightarrow \exists! e{:}{\scriptstyle M}.\mathit{Unit}(M,\circ,e)$
3. $\mathit{Group}(G,\circ) \Rightarrow \forall a{:}{\scriptstyle G}.\exists! x{:}{\scriptstyle G}.[a \circ x = e_G] \wedge [x \circ a = e_G]$
4. $\mathit{Group}(G,\circ) \Rightarrow \forall a{:}{\scriptstyle G}.\forall b{:}{\scriptstyle G}.\exists! c{:}{\scriptstyle G}.[a \circ b = c]$
5. $[\mathit{SubGroup}((U,\circ),(G,\circ)) \wedge \mathit{SubGroup}((V,\circ),(G,\circ))] \Rightarrow [e_G \in (U \cap V)]$
6. $[\mathit{Group}(G,\circ) \wedge \forall x{:}{\scriptstyle G}.(x \circ x) = e_G] \Rightarrow \mathit{Commu}(G,op)$

Table 5.2: Some simple theorems in group theory.

state that the unit element, the inverse of each element, and the divisors for two given elements are uniquely determined whenever an algebra has these properties. Theorem 4 expresses that for every two elements of a group the result of their multiplication is uniquely determined. The formulation of the theorems involve the quantifier of unique existence $\exists!$. For example, $\exists! e{:}{\scriptstyle G}.\mathit{Unit}(G,\circ,e)$ means there exists a unique element $e \in G$ such that e is the unit element of G. This is an abbreviation for the expression

$$\exists e{:}{\scriptstyle G}.\mathit{Unit}(G,\circ,e) \Rightarrow [\forall f{:}{\scriptstyle G}.\mathit{Unit}(G,\circ,f) \Rightarrow (e = f)]$$

The remaining two theorems are concerned with some simple consequences following from the definition of a group. Theorem 5 states that the unit element of a group is in the intersection of all of its *subgroups* and thus forms the smallest possible subgroup. The claim of theorem 6 is that all groups in which holds that $x \cdot x = e_G$ are *commutative*.

5.3 Formalization

In this section we formally define all the concepts necessary to formalize the theorems from the preceding section. In particular we shall be concerned with the formalization of properties of operations and, based on this, with the formal definition of algebraic structures.

5.3.1 Properties of Operations

We first give the formalizations for the group properties from definition 5.1. The first is the concept of a set being nonempty. The following definitions (5.2) to (5.5) are the straightforward formalizations of the properties $\mathcal{G}1$ to $\mathcal{G}4$ as given in definition 5.1. Thereby *Closed* and *Assoc* are binary predicates with a set $G_{\beta o}$ and an operation $\circ_{\beta\beta\beta}$ as arguments, whereas *Unit* and *Inverse* are ternary predicates, which in addition require an element of type β representing the actual unit element.

$$\mathit{NonEmpty} \equiv \lambda G_{\beta o}.\exists a.G(a) \tag{5.1}$$
$$\mathit{Closed} \equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\forall a_\beta{:}{\scriptstyle G}.\forall b_\beta{:}{\scriptstyle G}.G(a \circ b) \tag{5.2}$$
$$\mathit{Assoc} \equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\forall a_\beta{:}{\scriptstyle G}.\forall b_\beta{:}{\scriptstyle G}.\forall c_\beta{:}{\scriptstyle G}.(a \circ (b \circ c)) = ((a \circ b) \circ c) \tag{5.3}$$

$$\begin{aligned} \mathit{Unit} &\equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\lambda e_{\beta}.\forall a_{\beta}{:}G.[(a \circ e) = a] \wedge [(e \circ a) = a] && (5.4) \\ \mathit{Inverse} &\equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\lambda e_{\beta}.\forall a_{\beta}{:}G.\exists x_{\beta}{:}G.[(a \circ x) = e] \wedge [(x \circ a) = e] && (5.5) \end{aligned}$$

Analogous to their more complex counterparts in equations (5.4) and (5.5) we define the properties of the existence of a left unit element (property $\mathcal{G}3^*$) and of left inverses (property $\mathcal{G}4^*$) as two ternary predicates.

$$\begin{aligned} \mathit{LeftUnit} &\equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\lambda e_{\beta}.\forall a_{\beta}{:}G.(e \circ a) = a && (5.6) \\ \mathit{LeftInverse} &\equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\lambda e_{\beta}.\forall a_{\beta}{:}G.\exists x_{\beta}{:}G.(x \circ a) = e && (5.7) \end{aligned}$$

We also formalize property $\mathcal{G}3'$, the existence of divisors, as the binary predicate *Divisors*.

$$\mathit{Divisors} \equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\forall a_{\beta}{:}G.\forall b_{\beta}{:}G.[\exists x_{\beta}{:}G.(a \circ x) = b] \wedge [\exists y_{\beta}{:}G.(y \circ a) = b] \tag{5.8}$$

In addition to the properties we shall need to formalize the definitions of the various algebraic structures in section 5.3.2, we also define the property of commutativity of an operation in equation (5.9) as a binary predicate.

$$\mathit{Commu} \equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\forall a_{\beta}{:}G.\forall b_{\beta}{:}G.[(a \circ b) = (b \circ a)] \tag{5.9}$$

Finally, we give the definition of distributivity of two operations. Although we do not use it in this chapter we shall refer to it, albeit indirectly, in chapter 7. And thematically it fits well among the definitions presented in this section.

$$\begin{aligned} \mathit{Distrib} \equiv \lambda G_{\beta o}.\lambda \circ_{\beta\beta\beta}.\lambda \star_{\beta\beta\beta}.&\forall a_{\beta}{:}G.\forall b_{\beta}{:}G.\forall c_{\beta}{:}G. \\ &[(a \star (b \circ c)) = ((a \star b) \circ (a \star c))] \\ &\wedge[((a \circ b) \star c) = ((a \circ c) \star (b \circ c))] \end{aligned} \tag{5.10}$$

Distributivity is formalized with the ternary predicate *Distrib* that takes one set and two operations on this set as arguments. Furthermore, we define distributivity as both left and right distributivity.

5.3.2 Algebraic Structures

We shall define algebraic structures in terms of the properties of their set and their operation. For instance we define magma and semi-group as follows:

$$\begin{aligned} \mathit{Magma} &\equiv \lambda M_{\beta o}.\lambda \circ_{\beta\beta\beta}.\mathit{NonEmpty}(M) \wedge \mathit{Closed}(M,\circ) && (5.11) \\ \mathit{Semigroup} &\equiv \lambda S_{\beta o}.\lambda \circ_{\beta\beta\beta}.\mathit{NonEmpty}(S) \wedge \mathit{Closed}(S,\circ) \wedge \mathit{Assoc}(S,\circ) && (5.12) \end{aligned}$$

The formalization of semi-group in equation (5.12) differs from definition 5.3 in section 5.1 since we decided not to define our algebras in a hierarchical fashion. Thus, instead of defining a semi-group as an associative magma we rather give all of its properties immediately. This has the advantage that when using the definition while proving we do not have to recursively expand all the definitions of algebras before we arrive at the level of algebraic properties.

We next define the notion of a monoid as a composition of two elements, namely a set and an operation.

$$\begin{aligned} \mathit{Monoid} \equiv \lambda M_{\beta o}.\lambda \circ_{\beta\beta\beta}.&\mathit{NonEmpty}(M) \wedge \mathit{Closed}(M,\circ) \wedge \\ &\mathit{Assoc}(M,\circ) \wedge [\exists e_{\beta}{:}M.\mathit{Unit}(M,\circ,e)] \end{aligned} \tag{5.13}$$

Of course, we could have also defined a monoid with *LeftUnit* instead of *Unit*. However, we chose not to since in practical use it is more convenient to have both directions for the identity relation available instead of having to introduce an the appropriate theorem or even having to derive it anew every time.

The formalization of a monoid in equation (5.13) sticks as close as possible to the intuitive mathematical definition. But there is also the possibility to define a monoid as a triple consisting of a set, an operation, and the unit element. This would enable us to quantify over the unit element and refer to the this element in subsequent sub-formulas to the unit element by pulling these sub-formula in the scope of the quantifier, as we have already discussed in section 5.2. Thereby we could avoid the explicit reference, which leads to the use of the description operator as we shall see in the sequel. Although this might seem appealing on the first glance, this solution only postpones the general problem of coercion (i.e., how to lift elements from an underlying algebra into the newly defined algebra). Moreover, this treatment also decouples the occurrences of the unit element from its properties, which is not desirable in the context of proof planning as we shall discuss in chapter 6. For instance, we can consider a reformulation of the right hand side of theorem 3 in table 5.1 to $[\exists \star . \exists e . \mathit{Monoid}(G, \star, e) \wedge \mathit{Inverse}(G, \star, e)]$. Once the existential quantification of e is eliminated and the conjunction is split in a proof, it will be necessary to re-derive the unit property of the appropriate instantiation for e in any subproof the inverse property is involved.

Thus, before we formally define a group, we have to introduce the two reference expressions *StructUnit* and *LeftStructUnit* we have seen already in the theorems introduced in section 5.2.1.

$$\mathit{StructUnit} \equiv \lambda G_{\beta o} . \lambda \circ_{\beta\beta\beta} . \iota e_{\beta} . \mathit{Unit}(G, \circ, e) \tag{5.14}$$

$$\mathit{LeftStructUnit} \equiv \lambda G_{\beta o} . \lambda \circ_{\beta\beta\beta} . \iota e_{\beta} . \mathit{LeftUnit}(G, \circ, e) \tag{5.15}$$

Both predicates are used to refer to one particular element of the set G, namely the identity or the left identity, respectively. For instance, $\mathit{StructUnit}(G, \circ)$ refers to that unique element of G for which the unit property of equation (5.4) holds. We can now give the our formal definition of a *Group*:

$$\begin{aligned} \mathit{Group} \equiv \lambda G_{\beta o} . \lambda \circ_{\beta\beta\beta} . & \mathit{NonEmpty}(G) \wedge \mathit{Closed}(G, \circ) \\ & \wedge \mathit{Assoc}(G, \circ) \wedge [\exists e_{\beta}{:}_{G} . \mathit{Unit}(G, \circ, e)] \\ & \wedge \mathit{Inverse}(G, \circ, \mathit{StructUnit}(G, \circ)) \end{aligned} \tag{5.16}$$

Equation (5.16) demonstrates the advantage of using our formalization of the unit element with *StructUnit*. Although there are two explicit references to the unit element of the group structure, they can appear independent from each other, without being inside a common quantification.

Finally, we formalize the rest of the concepts needed for the theorems of section 5.2. We begin with the notions of a quasi-group and a loop.

$$\begin{aligned} \mathit{Quasigroup} \equiv \lambda Q_{\beta o} . \lambda \circ_{\beta\beta\beta} . & \mathit{NonEmpty}(Q) \\ & \wedge \mathit{Closed}(Q, \circ) \wedge \mathit{Divisors}(Q, \circ) \end{aligned} \tag{5.17}$$

$$\begin{aligned} \mathit{Loop} \equiv \lambda L_{\beta o} . \lambda \circ_{\beta\beta\beta} . & \mathit{NonEmpty}(L) \wedge \mathit{Closed}(L, \circ) \\ & \wedge \mathit{Divisors}(L, \circ) \wedge \exists e_{\beta}{:}_{G} . \mathit{Unit}(G, \circ, e) \end{aligned} \tag{5.18}$$

Another concept that occurs in the theorems of section 5.2 is that of a subgroup. Before we can define this, however, we need the formal notion of a subset:

$$\subseteq \equiv \lambda T_{\alpha o} . \lambda S_{\alpha o} . \forall x_{\alpha} . T(x) \Rightarrow S(x) \tag{5.19}$$

Equation (5.19) gives the traditional definition of a subset, stating that $T \subseteq S$ holds if all elements x of T are also elements of S. Having the subset definition available we can formalize that U is a subgroup of G as follows:

$$\begin{aligned} \mathit{SubGroup} \equiv \lambda U_{\beta o}\text{.}\lambda \star_{\beta\beta\beta}\text{.}\lambda G_{\beta o}\text{.}\lambda \circ_{b\beta\beta\beta}\text{.} \\ [\star = \circ] \wedge [U \subseteq G] \wedge [\mathit{Group}(U, \star)] \wedge [\mathit{Group}(G, \circ)] \end{aligned} \tag{5.20}$$

Thus, the *SubGroup* predicate takes four arguments: U, G, and their respective operations. We shall write this generally as $\mathit{SubGroup}((U, \star), (G, \circ))$ indicating that $(U, \star)$ is a subgroup of $(G, \circ)$. The predicate expresses that both structures have to have the same operation, U has to be a subset of G and both U and G have to be groups.

5.4 Generating Proofs Automatically

In this section we shall describe how we generate proofs with Ω-Ants automatically using the automation wrapper introduced in chapter 3.5. However, it is not feasible to use Ω-Ants automation with respect to all of Ωmega's inference rules or even with respect to the full calculus, as this leads to an intractable search problem. Therefore, we shall first introduce a goal directed search strategy for the first order fragment of our natural deduction calculus and then enhance this by inference rules for the treatment of both definitions and the description operator.

5.4.1 A Natural Deduction Search Procedure

In this section we shall introduce the natural deduction intercalation calculus Nic that allows one to search directly for normal proofs in the first order fragment of our natural deduction calculus. However, we shall give only a rough overview on the calculus and its search restrictions and refer to [55, 186] for more details.

The idea of Nic is to have an efficient, goal directed search procedure to derive first order normal proofs (i.e., cut-free proofs) in natural deduction. Therefore, the set of rules is strictly divided into introduction and elimination rules whose application is not only ordered but also restricted with respect to sub-formulas given in the premises. Thereby the search procedure relies on the notion of a *strictly positive sub-formula*, which we shall define following Byrnes in [55].

Definition 5.8 (Strictly Positive Sub-formula): Given any $C \in \{A \wedge B, B \wedge A, A \vee B, B \vee A, B \Rightarrow A\}$ with $A, B, C \in \mathbf{wff}_o(\Sigma)$ we write $A \lhd C$. We also write $A(t) \lhd \forall x\text{.}A(x)$ and $A(t) \lhd \exists x\text{.}A(x)$ for every term t. Let $\leq$ be the transitive and reflexive closure of $\lhd$. Whenever $A \leq C$ we say that A is a *strictly positive sub-formula* of C. □

Note that according to this definition $A \not\leq A \Rightarrow B$ but $\neg A \leq A \Rightarrow B$. Note also that the only strictly positive sub-formula of $\neg A$ is $\bot$ since it is equivalent to $A \Rightarrow \bot$.

5.4.1.1 Search Strategy

For the definition of an efficient search strategy we divide the rules of the first order fragment of Ωmega's calculus into two sets of elimination and introduction rules:

Elimination rules: $\neg_E$, $\wedge_{El}$, $\wedge_{Er}$, $\vee_E$, $\Rightarrow_E$, $\forall_E$, $\exists_E$, $\bot_E$

Introduction rules: $\neg_I$, $\wedge_I$, $\vee_{Il}$, $\vee_{Ir}$, $\Rightarrow_I$, $\forall_I$, $\exists_I$

In order to keep the search in the calculus goal directed we have certain restrictions on the applications of the above rules: Introduction rules can only be applied backward to an open goal. Elimination rules can only be applied when the current goal is a strictly positive sub-formula of one of its hypotheses. Then this sub-formula can be extracted using solely elimination rules. But on the contrary to Byrnes we do not yet fully support indirect proofs.

In addition to the restrictions on rule applications we also have restrictions on the order of their application: The application of an elimination rule is always preferred to the application of an introduction rule. This leads to the proof behavior that first a goal is decomposed with introduction rules until one of the emerging subgoals contains a formula that is a strictly positive sub-formula of one of the premises (*introduction phase*). From that point on elimination rules are applicable and are used to extract the necessary sub-formula to close the subgoal (*elimination phase*). Then again the introduction rules are used to decompose the next of the remaining subgoals.

Special restrictions apply for the application of elimination rules $\vee_E$ and $\exists_E$ since they can be applied at any point during the elimination phase provided the appropriate premises are given. However, their application can be restricted by using distinct search strategies.

$$\dfrac{A \vee B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}\,\vee_E \qquad\qquad \dfrac{\exists x. P(x) \quad \begin{matrix}P(t)\\ \vdots \\ Q\end{matrix}}{Q}\,\exists_E$$

The application of the $\exists_E$ rule is straightforward. It is applied exactly once to each occurring premise containing an existentially quantified formula. And it is applied immediately when the premise turns up for the first time in the proof, independent of whether we are in an elimination phase or not. Although this has the effect that some hypotheses might be derived during the proof that are actually not necessary, it has the advantage that the proof cannot fail because of the order of quantifier eliminations; that is, no witness term, which might have to depend on the term t, can be introduced before the $\exists_E$ rule is applied. The *eigenvariable* condition of the rule (i.e., the term t has to be new in the proof) also ensures that exactly one application of $\exists_E$ to a given premise is sufficient. Since t is always new in the proof the formula $P(t)$ cannot yet occur in any of the other premises and hence cannot be necessary for derivations within those premises. Thus, we do not need to derive multiple copies of the formula.

Note that we do not have to take similar precautions for the $\forall_I$ rule, because if we are in an introduction phase the universally quantified variable of the goal is substituted immediately, anyway. And in case we are in an elimination phase, the goal with the universal quantification is a strictly positive sub-formula of some premise and should therefore be derivable without eliminating the quantifier first.

Unfortunately, the application of the $\vee_E$ cannot be that easily restricted. Its application can be restricted neither to the case that the goal being a strictly positive sub-formula of a disjunctive premise nor to the case that the single disjuncts of a disjunctive premise are strictly positive sub-formulas in some of of the other given premises. This fact can be observed with a simple example: Suppose we want to proof S from a given set of premises $\Gamma = \{P \vee Q, P \Rightarrow R, Q \Rightarrow R, R \Rightarrow S\}$. Neither is S a strictly positive sub-formula of $P \vee Q$ nor are P or Q strictly positive sub-formulas of the remaining premises. Thus, we need a different criterion to restrict our search from branching with respect to all possible disjunctions at all new subgoals.

The criterion to restrict the application of $\vee_E$ is with respect to the overall search. For each disjunction in the premises we monitor whether a strictly positive sub-formula of one of the disjuncts occurs as subgoal during the proof search. If yes and if this proof branch is backtracked to a point beyond the occurrence of the subgoal, we monitor for the new proof attempt whether a subgoal occurs, which is a strictly positive sub-formula of the second disjunct. If this is also the case and if we have to backtrack again, the $\vee_E$ rule is applicable at the common origin of both backtracked proof attempts.

5.4.1.2 Quantifications and Unification

One of the main problems during automated proof search is how to determine the correct witness terms to instantiate variables. In automatic proof search this difficulty is usually avoided by using some kind of placeholder for a term and replace it with the actual term later, once we know what the correct term is. In proof planning, for example, this is done by introducing meta-variables, whose final value is constrained and eventually determined by a middle-out reasoning process. In more machine oriented calculi, such as resolution [181] or tableaux [192], the instantiation is postponed by introducing *free variables* and *Skolem functions*. *Unification* is then used to compute the appropriate instantiations.

Similarly, we can employ *Skolemization* and *unification* for automating the proof search in the first order variant of the NIC calculus. The respective free variables and Skolem functions are introduced with the quantifier rules.

In more detail, the application of the $\forall_E$ and $\exists_I$ rules replace the respective variables by new *free variables*[1], which does not yet occur elsewhere in the proof. As the dual operations in the application of the rules $\forall_I$ and $\exists_E$ the quantified variables are replaced with *Skolem functions*. Thereby a Skolem function is created from a new function symbol taking as its arguments all the free variables introduced into the proof so far. This ensures that the proper dependencies of the introduced term are respected, meaning that no free variable can be instantiated with a Skolem function that has been introduced into the proof after the variable, because this can violate *eigenvariable* conditions. In order to compute the correct instantiations for the variables unification is used to possibly make terms equal during rule application.

Since we work in a higher order setting, our implementation of the calculus differs from the original definition of BYRNES. In particular, we use higher order Skolem functions. (For an account on higher order Skolemization see [157].) We use also higher order unification to determine the instantiation of the variables. However, since higher order unification has some delicate properties such as being infinitary and undecidable (i.e., we never know how many higher order unifiers there are and whether the procedure to compute them will ever successfully terminate), we do not employ it in every rule application. Instead we introduce an adapted *Weaken* rule that tries to close a subgoal if it can find a unifiable counterpart in the premises. This concentrates the risk of employing higher order unification to a single inference rule, Moreover, the *Weaken* applicability always takes the first unifier, only, even if there exist several.

This way of treating higher order variables, Skolemization, and unification is rather *ad hoc* and does not correspond to a properly designed adaptation of NIC for higher order logic. How this extension can be achieved in a more refined way and also what properties an extended calculus would have has to be more thoroughly examined in the future. However, this is not subject of this thesis.

[1] In [55] BYRNES calls free variables "parameters". However, this naming would clash with our terminology introduced so far.

5.4.1.3 Modeling the Search in Ω-Ants

When modeling the proof search in Ω-Ants we have to essentially reflect the restriction criteria for the proof search and the ordering of the rule application. This can be done primarily with both the agents and their sorting heuristics.

Naturally, the restriction of the application directions in particular of the introduction rules is modeled by only allowing for those agents that guarantee the construction of the appropriate PAIs. Here we also include the search for the higher order unifier for the *Weaken* rule, such that in case the procedure does not terminate the agent can be reset by the resource agent.

The restriction on the applicability of elimination rules is achieved with a classification agent that provides the necessary information on the blackboards. If the classification agent detects that the subgoal currently considered is actually a strictly positive sub-formula of one of the premises it passes on appropriate information, thereby enabling agents for the elimination rules to run. In case the current open focus is closed the classification agent retracts the information.

The preference of rules is achieved with the sorting criterion of the suggestion agent, which prefers the *Weaken* rule before elimination and introduction rules. We do not have a criterion to sort suggestions for introduction rules since it cannot happen that two introduction rules are suggested at the same time. In case there is more than one applicable elimination rule we always prefer the one involving the 'youngest' nodes with respect to the chronological focus. This ensures that we concentrate on the decomposition of one formula and do not switch do another formula during this process.

The only restrictions we cannot model yet with the current components of the Ω-Ants architecture are those for the $\vee_E$ rule. They are implemented with the help of a hash-table that keeps track of all disjunctions in the premises, the occurrences of the subgoals that are strictly positive sub-formulas of the respective disjuncts, as well as of the appropriate backtracking points. The information can be accessed by the agents of $\vee_E$, which suggest the application of the rule accordingly.

5.4.2 Dealing with Description

The Nic calculus caters for the first order fragment of our calculus, only, with the exception of the higher order unification algorithm in the *Weaken* rule. However, some of our formalizations contain additionally the description operator, which cannot be dealt with by any of the Nic inference rules. Therefore, we extend the basic set of rules by two inference rules to handle description, which build on the rules to introduce or eliminate occurrences of ι:

$$\frac{\exists! x.\, P(x) \quad \forall z.\, P(z) \Rightarrow Q(z)}{Q\iota P}\ \iota_I \qquad \frac{Q\iota P \quad \begin{array}{c}[P(c)]\\ \vdots \\ \forall y.\, P(y) \Rightarrow (c = y)\end{array}}{\forall z.\, P(z) \Rightarrow Q(z)}\ \iota_E \text{ with } c \text{ new}$$

The ι_I tactic is based on the theorem that essentially states that if we know that there exists a unique element x for which $P(x)$ holds and P always implies Q for all possible z, we can infer that $Q\iota P$ holds.

$$\forall Q_{\beta o}.\, \forall P_{\beta o}.\, [[\exists! x_\beta.\, P(x)] \land [\forall z_\beta.\, P(z) \Rightarrow Q(z)]] \Rightarrow [Q\iota P]$$

Since the reverse direction of the above theorem does not necessarily hold, as with $Q\iota P$ we cannot assume that P actually uniquely describes an element, we have to base the ι_E tactic is based on a different theorem.

$$\forall Q_{\beta o}\textbf{.}\forall P_{\beta o}\textbf{.}\ [Q\iota P] \Rightarrow \\ [[\exists x_\beta\textbf{.}P(x) \Rightarrow \forall y_\beta\textbf{.}P(y) \Rightarrow (x = y)] \Rightarrow [\forall z_\beta\textbf{.}P(z) \Rightarrow Q(z)]]$$

The theorem states that from both $Q\iota P$ and the existence of a unique element in P we can deduce that $\forall z_\beta\textbf{.}P(z) \Rightarrow Q(z)$ holds. Since the term c essentially stems from an elimination of an existentially quantified variable it has to be new. During automatic proof search it is instantiated with a new Skolem function.

The ι_I and ι_E tactics deal with single occurrences of the description operator in formulas. But during the actual proof search we use the more complex tactics ι_I^* and ι_E^*, which can eliminate all occurrences of the description operator in a formula within a single step. Both tactics are essentially iterative applications of the simpler ι_I and ι_E tactics given above. Since both tactics are rather procedural we omit their declarative presentation. For instance, the ι_I^* tactic applied to an open goal containing n occurrences of the description operator leads to an elimination of the n occurrences starting with the innermost. This results in $n + 1$ new open goals, where the first n are the appropriate unique existence statements and the last new open goal is a nested implication corresponding to applying the ι_I tactics n times.

The tactics dealing with description are applied only if no other rule is applicable anymore and in the order of always applying ι_E^* before ι_I^*. This has the advantage that their application can be postponed until the last possible moment, which keeps the proof search leaner and the formulas more compact. Moreover, it ensures that all necessary hypotheses, for instance created by the application of the $\Rightarrow_I$ rule, that are vital for successfully proving the subgoals produced by the ι_I^* and ι_E^* tactics are already available.

5.4.3 Dealing with Definitions

In the Nic calculus all occurring constants are treated as primitive symbols. However, our problem formulations contain defined concepts such as *Group* etc. In order to deal with them we use variations of the ΩMEGA rules $\equiv_I$ and $\equiv_E$ for introducing and eliminating definitions we have discussed in chapter 2.2.2.

$$\frac{A}{[t'/t]B}\ \equiv_E^*(t\ \equiv\ t') \qquad \frac{[t'/t]A}{B}\ \equiv_I^*(t\ \equiv\ t')$$

The difference to the rules from chapter 2.2.2 is that $\equiv_E^*$ and $\equiv_I^*$ expand all occurrences of a definition $t\ \equiv\ t'$ in a given formula in one step. It therefore does not need the position of the definition as a parameter. The expansion of these two tactics is then the iterative application of either the $\equiv_I$ rule or the $\equiv_E$ rule.

Possible definition expansions are always preferred to the application of the Nic calculus rules except for those rules dealing with quantifiers and the *Weaken* rule, which is always promoted over all other possibly applicable inference rules. The expansion of definitions can be further restricted by specifying a list of concepts that are never to be expanded. For our examples we shall not allow the expansion of equality $=$ and the sorted quantifiers. Despite this restriction our treatment of definition expansion is rather ad hoc.

If the problem formulations are always given as single theorems (i.e., there is an initial conclusion and no explicit hypotheses), only the $\equiv_I^*$ rule is actually necessary. But in the case where formulas of hypotheses are explicitly given we also need the $\equiv_E^*$ rule.

5.4.4 Adding Automated Theorem Provers

In addition to the rules of the NIC calculus and the inference rules for dealing with definitions and description we also enrich our automatic proof search by computations of automated theorem provers. In particular we use the first order prover OTTER and the higher order provers TPS and LEO. In the examples we are dealing with in this chapter the single provers are applied to completely solve given subproblems alone. However, our architecture also allows for the cooperation of different theorem provers, for instance in a way that LEO only partially solves a problem, but returns a set of first order clauses, which can then be successfully refuted by OTTER. For examples of fruitful cooperations between the first order and the higher order theorem provers see [23, 24]. The use of TPS and LEO ensures that search specialized for a higher order context is performed. Albeit, this is not always fruitful as we shall further discuss in section 5.6.

The ATPs are incorporated into Ω-ANTS by testing whether their corresponding commands are applicable; that is, we have agents that check the possible applicability of a certain prover to a given subproblem and agents that run it in a background process. When the prover has produced a proof for a given subgoal, its application is always preferred to any of the other inference rules, apart from *Weaken*. In case an ATP does not produce a result within the given time bound of the automation wrapper it is stopped by the resource agent. Compare the algorithm in chapter 3.5.2 table 3.5.

Depending on the given time bound the behavior of the ATPs involved can change; that is, the larger the time bound the earlier the ATPs can sometimes solve the problem. Therefore, depending on the time bound the appearance of the final proofs can also change.

5.5 Example

As an example consider the automatically generated proof for theorem 1 in table 5.1 that is given in table 5.3. To keep things concise we have only given those parts of the proof we actually focus on during the explanation and also abbreviated some of the formulas and hypotheses lists.

The first step in the proof is the application of $\forall_I$ to the actual theorem given in line L_1. This introduces the first *Skolem constant* sk_0 (i.e., a Skolem function with zero arity) as no free variable has been introduced so far. Since no further NIC rules can be applied to the resulting line L_2 Ω-ANTS starts with a series of definition expansions, until even the equivalence connective has been expanded in line L_{13}. Observe, the particularity of the $\equiv_I^*$ rule for instance in line L_3 that eliminates two occurrences of *NonEmpty* in a single step.

In line L_{13} are for the first time NIC rules applicable again. This leads to the split of the conjunction and two new open subgoals, namely L_{14} and L_{15}. Both subproofs are basically analogous and we shall concentrate on the latter. The first step here is the $\Rightarrow_I$ application, which also leads to the first premise in our proof. Since the premise contains an existentially quantified formula the $\exists_E$ rule is immediately applied to line L_{16} introducing sk_1 as another Skolem constant[2]. After that, however, no more elimination steps are carried out since our goal L_{19} is not a strictly positive sub-formula of the premises. Instead Ω-ANTS further decomposes the given goal by applying $\exists_I$ and $\wedge_I$. The former introduces a new free variable

[2] For better readability we write the resulting function in prefix instead of infix notation (i.e., $sk_1(a, b)$ instead of $a\, sk_0\, b$).

$L_{16}.$	L_{16}	$\vdash \exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \star b)] \wedge \ldots$	(Hyp)
$L_{18}.$	L_{18}	$\vdash [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(sk_1(a,b))]$ $\wedge[\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot \forall c{:}sk_0 \centerdot sk_1(a, sk_1(b,c)) = sk_1(sk_1(a,b),c)]$ $\wedge[\exists e{:}sk_0 \centerdot \forall a{:}sk_0 \centerdot sk_1(x,a) = a]$ $\wedge[\forall a{:}sk_0 \centerdot \exists x{:}sk_0 \centerdot sk_1(x,a) = \iota x \centerdot \forall a{:}sk_0 \centerdot sk_1(x,a) = a]$	(Hyp)
$L_{29}.$	L_{29}	$\vdash \forall a{:}sk_0 \centerdot sk_1(c_1, a) = a$	(Hyp)
$L_{30}.$	$\mathcal{H}_2$	$\vdash \forall x \centerdot \forall a{:}sk_0 \centerdot [sk_1(x,a) = a] \Rightarrow [x = c_1]$	$(Otter\ L_{29})$
$L_{31}.$	L_{18}	$\vdash \forall y \centerdot [\forall a{:}sk_0 \centerdot sk_1(y,a) = a] \Rightarrow$ $[\exists a \centerdot sk_0(a) \wedge \forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(sk_1(a,b))$ $\wedge[\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot \forall c{:}sk_0 \centerdot sk_1(a, sk_1(b,c)) = sk_1(sk_1(a,b),c)]$ $\wedge[\exists e{:}sk_0 \centerdot \forall a{:}G \centerdot sk_1(e,a) = a] \wedge [\forall a{:}sk_0 \centerdot \exists x{:}sk_0 \centerdot sk_1(x,a) = y]]]$	$(\iota^*_E\ L_{18}\ L_{30})$
$L_{36}.$	$\mathcal{H}_1$	$\vdash \exists y \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,y) = a] \wedge [var_0(y,a) = a]] \wedge$ $[\forall z \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,z) = a] \wedge [var_0(z,a) = a]] \Rightarrow [z = y]]$	$(Otter$ $L_{18}\ L_{31})$
$L_{34}.$	$\mathcal{H}_1$	$\vdash \exists! z \centerdot \forall a{:}sk_0 \centerdot [var_0(a,z) = a] \wedge [var_0(z,a) = a]$	$(\equiv^*_I\ L_{36}\ \exists!)$
$L_{35}.$	$\mathcal{H}_1$	$\vdash \exists y \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,y) = a] \wedge [var_0(y,a) = a]] \wedge$ $[\forall z \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,z) = a] \wedge [var_0(z,a) = a]] \Rightarrow [z = y]]$	$(Otter$ $L_{18}\ L_{31})$
$L_{33}.$	$\mathcal{H}_1$	$\vdash \exists! y \centerdot \forall a{:}sk_0 \centerdot [var_0(a,y) = a] \wedge [var_0(y,a) = a]$	$(\equiv^*_I\ L_{35}\ \exists!)$
$L_{32}.$	$\mathcal{H}_1$	$\vdash \forall y \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,y) = a] \wedge [var_0(y,a) = a]] \Rightarrow$ $[\forall z \centerdot [\forall a{:}sk_0 \centerdot [var_0(a,z) = a] \wedge [var_0(z,a) = a]] \Rightarrow$ $[\forall a{:}sk_0 \centerdot \exists x{:}sk_0 \centerdot var_0(a,x) = y \wedge var_0(x,a) = z]]$	$(Otter$ $L_{18}\ L_{31})$
$L_{28}.$	$\mathcal{H}_1$	$\vdash [\forall a{:}sk_0 \centerdot \exists x{:}sk_0 \centerdot$ $[var_0(a,x) = \iota x \centerdot \forall b{:}sk_0 \centerdot [var_0(b,x) = b] \wedge [var_0(x,b) = b]]$ $\wedge[var_0(x,a) = \iota x \centerdot \forall b{:}sk_0 \centerdot [var_0(b,x) = b] \wedge [var_0(x,b) = b]]]$	$(\iota^*_I\ L_{32}$ $L_{33}\ L_{34})$
		$\vdots$	
$L_{23}.$	$\mathcal{H}_1$	$\vdash [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(var_0(a,b))]_{\{var_0 \leftarrow sk_1\}}$	$(Leo\ L_{18})$
$L_{22}.$	$\mathcal{H}_1$	$\vdash [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(var_0(a,b))] \wedge \ldots \wedge$ $[\forall a{:}sk_0 \centerdot \exists x{:}sk_0 \centerdot$ $[var_0(a,x) = \iota x \centerdot \forall b{:}sk_0 \centerdot [var_0(b,x) = b] \wedge [var_0(x,b) = b]]$ $\wedge[var_0(x,a) = \iota x \centerdot \forall b{:}sk_0 \centerdot [var_0(b,x) = b] \wedge [var_0(x,b) = b]]]$	$(\wedge_I\ L_{23}\ L_{24})$
$L_{21}.$	$\mathcal{H}_1$	$\vdash [\exists a \centerdot sk_0(a)]$	$(Otter\ L_{18})$
$L_{20}.$	$\mathcal{H}_1$	$\vdash [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(var_0(a,b))] \wedge \ldots$	$(\wedge_I\ L_{19}\ L_{20})$
$L_{19}.$	$\mathcal{H}_1$	$\vdash \exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots$	$(\exists_I\ L_{20}\ var_0)$
$L_{17}.$	L_{16}	$\vdash \exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots$	$(\exists_E\ L_{16}\ L_{19})$
$L_{15}.$		$\vdash [\exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \star b)] \wedge \ldots] \Rightarrow$ $[\exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots]$	$(\Rightarrow_I\ L_{16})$
		$\vdots$	
$L_{14}.$		$\vdash [\exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots] \Rightarrow$ $[\exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \star b)] \wedge \ldots]$	$(\Rightarrow_I\ \ldots)$
$L_{13}.$		$\vdash [[\exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots] \Rightarrow$ $[\exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \star b)] \wedge \ldots]]$ $\wedge[[\exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \star b)] \wedge \ldots] \Rightarrow$ $[\exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge [\forall a{:}sk_0 \centerdot \forall b{:}sk_0 \centerdot sk_0(a \circ b)] \wedge \ldots]]$	$(\wedge_I\ L_{14}\ L_{15})$
		$\vdots$	
$L_4.$		$\vdash [\exists \circ \centerdot [\exists a \centerdot sk_0(a)] \wedge Closed(sk_0, \circ) \wedge Assoc(sk_0, \circ) \wedge$ $[\exists e_\beta{:}sk_0 \centerdot Unit(sk_0, \circ)] \wedge Inverse(sk_0, \circ, StructUnit(sk_0, \circ))]$ $\Leftrightarrow [\exists \star \centerdot [\exists a \centerdot sk_0(a)] \wedge Closed(sk_0, \star)$ $\wedge Assoc(sk_0, \star) \wedge [\exists e{:}sk_0 \centerdot LeftUnit(sk_0, \star, e)]$ $\wedge LeftInverse(sk_0, \star, LeftStructUnit(sk_0, \star))]$	$(\equiv^*_I\ L_5$ $Closed)$
$L_3.$		$\vdash [\exists \circ \centerdot NonEmpty(sk_0) \wedge Closed(sk_0, \circ) \wedge Assoc(sk_0, \circ) \wedge$ $[\exists e_\beta{:}sk_0 \centerdot Unit(sk_0, \circ, e)] \wedge Inverse(sk_0, \circ, StructUnit(sk_0, \circ))]$ $\Leftrightarrow [\exists \star \centerdot NonEmpty(sk_0) \wedge Closed(sk_0, \star)$ $\wedge Assoc(sk_0, \star) \wedge [\exists e{:}sk_0 \centerdot LeftUnit(sk_0, \star, e)]$ $\wedge LeftInverse(sk_0, \star, LeftStructUnit(sk_0, \star))]$	$(\equiv^*_I\ L_4$ $NonEmpty)$
$L_2.$		$\vdash [\exists \circ \centerdot Group(sk_0, \circ)] \Leftrightarrow [\exists \star \centerdot NonEmpty(sk_0) \wedge Closed(sk_0, \star)$ $\wedge Assoc(sk_0, \star) \wedge [\exists e{:}sk_0 \centerdot LeftUnit(sk_0, \star)]$ $\wedge LeftInverse(sk_0, \star, LeftStructUnit(sk_0, \star))]$	$(\equiv^*_I\ L_3$ $Group)$
$L_1.$		$\vdash \forall G \centerdot [\exists \circ \centerdot Group(G, \circ)] \Leftrightarrow [\exists \star \centerdot NonEmpty(G) \wedge Closed(G, \star)$ $\wedge Assoc(G, \star) \wedge [\exists e{:}G \centerdot LeftUnit(G, \star, e)]$ $\wedge LeftInverse(G, \star, LeftStructUnit(G, \star))]$	$(\forall_I\ L_2\ sk_0)$

$\mathcal{H}_1 = \{L_{16}, L_{18}\}$ $\mathcal{H}_2 = \{L_{18}, L_{29}\}$ $\mathcal{H}_3 = \{L_{18}, L_{31}\}$

Table 5.3: An automatically generated equivalence proof.

var_0. We can also observe that due to the order of applying $\exists_E$ and $\exists_I$ sk_1 does not depend on the variable var_0. This is crucial since var_0 has to be unified with sk_1 later on.

At this point we have for the first time a goal (L_{21}) that is a strictly positive sub-formula of the premise L_{18}. Therefore, Ω-Ants can now start decomposing the premise with Nic elimination rules. However, we have also an automated theorem prover, namely Otter that can immediately derive the goal. This is possible since in this case no higher order variable occurs, which Otter can of course not deal with.

This is different for the next subgoal L_{23}, which we get after further decomposition. Here the higher order variable var_0, which Otter cannot unify with the appropriate term in the premise. But the problem can be easily solved by Leo, which also returns the appropriate unifier $\{var_0 \leftarrow sk_1\}$ which is indicated as the subscript of line L_{23}. The substitutions are not carried out immediately in the overall proof but merely added as a constraint for the particular free variable. The substitution is taken into account, however, for the next unification or when occurrences of the variable are passed to an automated theorem prover. This treatment of substitutions eases backtracking and is similar to the treatment of meta-variables, which we shall discuss in more detail in the subsequent chapters. Therefore, the next properties, associativity and unit element, can be shown using Otter again since var_0 is replaced with sk_1 before the prover is called. To preserve some space we have omitted these parts of the proof.

This leaves only the property of inverses to show, which is more difficult because it can neither be immediately inferred from the premises nor successfully shown by one of the automated theorems provers since the formula involves the description operator. Their are also no more definitions that can be expanded since neither the sorted quantifiers nor the equality are considered by $\equiv^*_I$, and there is no applicable Nic rule, since there is none for the treatment of the sorted exists quantifier. Thus, at this point in the subproof the description operators occurring in lines L_{18} and L_{28} are eliminated. Note that the order in which the tactics are applied, namely ι^*_E before ι^*_I, is important. Because if ι^*_I would have been applied first, other introduction rules would have been applicable delaying the application of ι^*_E by a long fruitless search.

The elimination of the description operator in line L_{18} leads to a new premise for the subproof, line L_{31} and one new open goal, line L_{30}, which is supported by the newly introduced hypothesis L_{29}. This new subgoal is immediately derived by Otter from the new hypothesis. The application of ι^*_I to line L_{28} yields three new open subgoals, namely L_{32}, L_{33}, and L_{34}. L_{32} contains the nested implication, which can be directly shown by Otter. Lines L_{33} and L_{34} contain the uniqueness property of the unit element stated with the $\exists!$ quantifier. Since this is a defined concept it has to be expanded before Otter can successfully prove the statements.

In our example it is obvious that certain expansions of definitions are superfluous or at least were carried out too soon in the proof. For instance the concepts *NonEmpty*, *Closed*, and *Assoc*, which appear on both sides of the equivalence could be immediately shown without expanding them first. Nevertheless the expansion of *Assoc* is still necessary later on in the proof since associativity is important for showing the existence of inverses from the existence of left inverses. Thus, the proof could be both shortened and simplified by using a clever strategy for expanding definitions.

An ad hoc strategy would be, for instance, to check whether a certain defined concept appears on both sides of an equivalence or an implication. A more elaborate solution is, for instance, to implement a dual instantiation strategy as introduced by Bishop into the mating search of tps [35]. Thereby the proof search is split into two branches for each occurring defined concept where in one branch the concept's definition is left unexpanded and in the other branch the concept is replaced by its definition. However, this would call for introducing *or* parallelism for more than a single proof step into the search, which the central proof object, the $\mathcal{PDS}$, does not yet permit. The investigation of these types of enhancements will be subject of future work.

All our example theorems are essentially proved in the same fashion: The Ω-Ants mechanism divides the problem into chunks that can be solved by an automatic theorem prover. Thereby only the theorems from table 5.2 can be solved using Otter alone, while all others involve higher order unification, which either has to be done within the *Weaken* rule or by either Leo or tps. If the higher order variables are sufficiently constrained we can then apply Otter again. However, all this might not be enough as we shall examine in the next section.

5.6 A Challenging Problem

In this section we discuss an example of an equivalence proof that is trivial from a mathematical point of view, however, very challenging from an automated reasoning viewpoint. It is constructed using an alternative definition of a group taken from [104]. The peculiarity of this definition is that it introduces inverses with a distinct unary operation.

Definition 5.9 (Group — alternative): Let G be a nonempty set. G is a *group* if the following holds:

$\mathcal{H}1$) For each two $a, b \in G$ is a binary product $\cdot$ defined such that $a \cdot b = c$ with $c \in G$ uniquely defined.

$\mathcal{H}2$) We have unary operation inverse $^{-1}$ such that for each $a \in G$ the inverse $a^{-1} \in G$ is uniquely determined.

$\mathcal{H}3$) For all $a, b, c \in G$ holds $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

$\mathcal{H}4$) For all $a, b \in G$ holds $a^{-1} \cdot (a \cdot b) = b = (b \cdot a) \cdot a^{-1}$.

□

Informally it is trivial to show that definitions 5.1 and 5.9 are indeed equivalent. The properties $\mathcal{H}1$, $\mathcal{H}2$, and $\mathcal{H}3$ can be easily identified with $\mathcal{G}1$, $\mathcal{G}4$, and $\mathcal{G}2$ respectively. The required uniqueness properties in $\mathcal{H}1$ and $\mathcal{H}3$ can be inferred, for instance, with the help of the theorems 4 and 3. Identifying $\mathcal{H}2$ and $\mathcal{G}4$ is also facilitated by the simple insight that the inverse function of definition 5.9 is nothing but a different notation for the inverse elements. The only slightly tricky part is to derive the existence of a unit element ($\mathcal{G}3$) since we have to identify the identity e with $a \cdot a^{-1}$ for any given $a \in G$. Having this equality available to show that $\mathcal{H}4$ holds in the opposite direction is straightforward.

If we now painstakingly formalize the problem the proof becomes far more difficult. We start by giving the formal definitions for the properties $\mathcal{H}1$, $\mathcal{H}2$, and $\mathcal{H}4$ in the equations (5.21), (5.22), and (5.23), respectively.

$$\mathit{ClosedUnique} \equiv \lambda G_{\beta o} \mathbf{.} \lambda \circ_{\beta\beta\beta} \mathbf{.} \forall a_\beta{:}G \mathbf{.} \forall b_\beta{:}G \mathbf{.} \exists! c_\beta{:}G \mathbf{.} (a \circ b) = c \tag{5.21}$$

$$\mathit{UniqueInv} \equiv \lambda G_{\beta o} \mathbf{.} \lambda {}^{-1}{}_{\beta\beta} \mathbf{.} \forall a_\beta{:}G \mathbf{.} \exists! b_\beta{:}G \mathbf{.} b = a^{-1} \tag{5.22}$$

$$\begin{aligned} \mathit{InvLaw} \equiv{} & \lambda G_{\beta o} \mathbf{.} \lambda \circ_{\beta\beta\beta} \mathbf{.} \lambda {}^{-1}{}_{\beta\beta} \mathbf{.} \\ & \forall a_\beta{:}G \mathbf{.} \forall b_\beta{:}G \mathbf{.} [(a^{-1} \circ (a \circ b)) = b] \wedge [((b \circ a) \circ a^{-1}) = b] \end{aligned} \tag{5.23}$$

The above predicates are concerned with two different operations on the given set: The binary group operation and the unary inverse operation. Therefore, the group definition involving these predicates also has to speak about two operations instead of just one as the *Group* predicate defined in (5.16).

Hence, when actually formulating the equivalence theorem we have to introduce a variable for the inverse operation as well.

$$\begin{aligned}[\exists \circ_{\bullet} Group(G,\circ)] \Leftrightarrow [\exists \star_{\bullet} NonEmpty(G) \wedge ClosedUnique(G,\star) \wedge Assoc(G,\star)\\ \wedge[\exists {}^{-1}{}_{\bullet} UniqueInv(G,{}^{-1}) \wedge InvLaw(G,\star,{}^{-1})]]\end{aligned}$$

Here the choice of the position for the existential quantification involving $^{-1}$ is only to ease interactive proof construction. We could have also quantified the variable over the whole conjunction.

Although parts of the proof can be easily solved by Leo, proving the whole theorem automatically with Ω-Ants fails: At some point in the proof the existential quantification for the inverse operation has to be eliminated. Depending on the direction of the equivalence that has to be shown Ω-Ants either inserts a Skolem function or a variable. However, neither Ω-Ants itself nor TPS nor Leo are able to compute an appropriate unifier to replace the variable. The actual instantiation is of the form:

$$\lambda a_{\bullet} \iota x_{\bullet} [a \circ x = StructUnit(G,\circ)] \wedge [x \circ a = StructUnit(G,\circ)] \tag{5.24}$$

This lambda term denotes a unary function that returns for each element a the unique x that satisfies the inverse property as given in definition 5.1. This corresponds to a precise formalization of the step in the informal proof where we simply equate the inverse function with the inverse elements. Naturally, it is also sufficient to only have the right or the left inverse equation in the lambda expression.

But even then the computation of this term is beyond the scope of any contemporary automated theorem provers and their respective unification algorithms. The construction of the term requires *primitive substitution* [3] involving description during the unification. However neither TPS nor Leo can currently successfully solve this problem. Therefore, both these two agents and the *Otter* agent fail in their proof attempts and hence we cannot automate the search for the proof of our equivalence statement with Ω-Ants. However, we can conduct the complete proof interactively in ΩMEGA with support from Ω-Ants when instantiating the appropriate term by hand.

So why do we fail to automate the proof although it is actually trivial from a mathematical point of view? One point of critique could be to blame our formalization. So far we have always chosen a formalization that is as close to the mathematical formulation (of the properties, the algebraic structures, or the proofs) as possible. But maybe in this case a reformulation of both the properties and the problem might turn out to be useful.

Let us first take into account how problems in group theory are formalized for first order theorem provers, for instance, in the TPTP [197, 198] library: There all formalization of the group axioms introduce an explicit identity and an inverse function, which has defined properties similar to property $\mathcal{G}4$. This means the knowledge on how to handle the connection between inverse elements and the inverse operation is a priori compiled into the axiomatization. Naturally, we cannot explore the full implications of our theorem in this encoding anyway since there is no way of equating functions and thus the actual plot is lost in the first place. Of course, one could encode in Zermelo-Fränkel set theory but this would certainly not contribute to a better automation.

Thus, if we look at the problem again from a higher order perspective, then there are two easy ways to fix our formalization to make the problem go through smoothly: Either we enrich the first group definition by introducing an explicit inverse operation connected to property $\mathcal{G}4$. Or we define the inverse operation in the theorem similar for the unit element by introducing a predicate of type $\beta \rightarrow \beta$ that, when expanded, spells out the property $\mathcal{G}4$ for the resulting inverse element. The predicate itself is essentially of the same form as the lambda expression in (5.24). Although with either rectification the theorem can easily be solved automatically, we completely obscured the inventive part of the proof since we encoded basically the solution already in the problem description. Thus, the only challenging intellectual part of the proof cannot be done yet with an automaton.

In contrast, as humans we easily solve the problem mainly for two reasons: Firstly, we can relate the properties to each other on an abstract level, figure out which of the properties are nearly the same, and see therefore easily how the inverse operation and the unit element have to be instantiated. Secondly, the high level reasoning frees us from worrying how exactly we have to identify the inverse function in one definition with the existence of inverse elements in the other.

In summary we can conclude that if we want to solve the problem in the given form, we have to use higher order logic, since first order logic does not allow us to state the problem properly. In higher order logic the only mechanism that could derive a solution for our instantiation problem is primitive substitution [3], for which it remains to be seen whether it will ever become tractable for term constructs involving connectives and quantification, let alone description. For capturing the human intuition behind the proof it might not even be useful to adhere strictly to the logic level. This, however, raises the question: What actually is an adequate way of capturing both mathematical precision and intuition without forgoing correctness?

5.7 Summary of Chapter 5

In this chapter we have presented goal directed automatic proof search with the Ω-Ants mechanism by modeling the Nic calculus and its restrictions on search and order of rule application. With the slight extensions necessary to handle the restrictions for the $\vee_E$ rule, the Nic's search heuristics can be modeled without changing Ω-Ants. The calculus can be adapted to our needs by enriching it with higher order Skolemization and unification as well as by adding inference rules that can deal with definitions and description. Furthermore, we have added automated theorem provers that can tackle whole subproblems autonomously. However, their success depends on the time bound given in the automation wrapper of Ω-Ants.

We have successfully applied this prover to a small number of simple example theorems. Here the approach of the system is essentially to decompose a given problem into chunks that can be easily solved by one of the connected automated theorem provers. The resulting proofs can vary with respect to the given time bound and the speed of an automated theorem prover to find a proof. The theorems involved where mainly equivalence statements and uniqueness statements, which are, from a mathematical point of view, relatively trivial. However, this does not necessarily mean that they are trivial from a logical and theorem proving point of view as we have seen with the concluding challenging problem. Here we had again a mathematically trivial equivalence theorem, which cannot yet be solved automatically with current theorem proving techniques.

Chapter 6

Homomorphism Theorems

In this chapter we present a case study for the interactive proof planning approach we have presented in section 4.1.1. We shall illustrate this approach with the help of proofs for a class of theorems for homomorphism statements in group theory. We use Ω-Ants as an algorithm for Multi and parameterize the algorithm using planning strategies. Since the proofs we are dealing with are all of a similar scheme we implement a single strategy using both domain specific and domain independent methods. Proof planning is then done either interactively or automatically using the Ω-Ants mechanism.

We shall first introduce the problems and give the formalization for concepts we require in addition to those already defined in the preceding chapter. We then present how the proofs for homomorphism theorems are interactively planned in general before elaborating the scheme with a concrete example.

6.1 Homomorphism Problems

The problems we are concerned with in this case study, are essentially derived from six major theorems involving the homomorphism property. The actual problems are either the theorems themselves, weaker version of the theorems, or the different lemmas needed to prove them. Table 6.1 introduces the six theorems.

The theorems 1 to 4 and 6 state some properties of a homomorphism f between two given groups G and H, whereas theorem 5 is concerned with homomorphism properties between three groups involved. In detail, theorems 1 and 3 state that the image and the kernel of f are subgroups in H. In case G is commutative the image of f is also commutative with the operation of H (theorem 2); if the kernel of f consists only of the neutral element of G, f itself is injective (theorem 4). Theorem 5 reads that given three groups G, H, K and two homomorphisms of the form $f_1 : G \rightarrow H$ and $f_2 : H \rightarrow K$, then the composition $f_2 \bullet f_1$ is again a homomorphism from G to K. Here $\bullet$ denotes the composition of mappings, such that for a $g \in G$ holds $f_2(f_1(g)) \in K$.

Theorem 6 is the most difficult of our homomorphism problems and is taken from [77]. It states that if we have two groups G, H and a surjective homomorphism $f_1 : G \rightarrow H$ and if we have an additional homomorphism f_2 from G into some arbitrary structure $(K, \diamond)$ and a mapping $\varphi : H \rightarrow K$, such that $f_2(x) = \varphi(f_1(x))$ for all $x \in G$, then φ is also a homomorphism.

For the latter theorem we can also prove two weaker versions if we do not assume

1. $$[Group(G,\circ) \wedge Group(H,\star) \wedge Hom(f,(G,\circ),(H,\star))] \\ \Rightarrow [SubGroup(Im(f,G),(H,\star))]$$

2. $$[Group(G,\circ) \wedge Group(H,\star) \wedge \\ Hom(f,(G,\circ),(H,\star)) \wedge Commu(G,\circ)] \Rightarrow [Commu(Im(f,G),\star)]$$

3. $$[Group(G,\circ) \wedge Group(H,\star) \wedge Hom(f,(G,\circ),(H,\star))] \\ \Rightarrow [SubGroup((Kern(f,G,e_H),\circ),(H,\star))]$$

4. $$[Group(G,\circ) \wedge Group(H,\star) \wedge \\ Hom(f,(G,\circ),(H,\star)) \wedge [Kern(f,G,e_H) = e_G]] \Rightarrow [Inj(f,G)]$$

5. $$[Group(G,\circ) \wedge Group(H,\star) \wedge Group(K,\diamond) \wedge \\ Hom(f_1,(G,\circ),(H,\star)) \wedge Hom(f_2,(H,\star),(K,\diamond)) \\ \Rightarrow Hom(f_2 \bullet f_1,(G,\circ),(K,\diamond))$$

6. $$[Group(G,\circ) \wedge Group(H,\star) \wedge Hom(f_1,(G,\circ),(H,\star)) \wedge Surj(f_1,G,H) \\ \wedge Hom(f_2,(G,\circ),(K,\diamond)) \wedge [\forall x{:}G.\, f_2(x) = \varphi(f_1(x))]] \\ \Rightarrow [Hom(\varphi,(H,\circ),(K,\diamond))]$$

Table 6.1: The homomorphism theorems.

that f_1 is a surjection. We can then show that φ is a homomorphism from the image of f_1 into K as well as from the image of f_1 into the image of f_2.

In theorems 1 and 3 we have to show that the image and the kernel of the homomorphism f are subgroups of the group H in the codomain of f. Thus, we not only have to show that the image and kernel are subsets of H but also that the group properties hold. We have realized this by splitting the proof of the actual theorem into a set of lemmas; one for each group property. For instance, one lemma for theorem 1 is

$$[Group(G,\circ) \wedge Group(H,\star) \wedge Hom(f,(G,\circ),(H,\star))] \Rightarrow [Assoc(Im(f,G),\star)]$$

Depending on the axiomatization of a group we can derive different sets of lemmas and therefore gain a variety of different homomorphism theorems.

6.2 Formalization

In order to formalize and prove the theorems from the preceding section we have to introduce some more concepts in addition to those already defined in chapter 5. We start with the most important notion for the theorems in this chapter, the concept of a homomorphism.

$$Hom \equiv \lambda h_{\alpha\beta}.\lambda A_{\alpha o}.\lambda \circ_{\alpha\alpha\alpha}.\lambda B_{\beta o}.\lambda \star_{\beta\beta\beta}. \\ \forall x_\alpha{:}A.\forall y_\alpha{:}A.\, h(x \circ y) = h(x) \star h(y) \qquad (6.1)$$

The definition states that a function h, which maps elements of type α to elements of type β is a homomorphism between the two structures $(A,\circ)$ and $(B,\star)$ if for all elements $x, y \in A$ $h(x \circ y) = h(x) \star h(y)$ holds.

Two other functional properties we need for the theorems given above are the concepts injective and surjective. These are again defined straightforwardly:

$$Inj \equiv \lambda f_{\alpha\beta}.\lambda A_{\alpha o}.\forall x_\alpha{:}A.\forall y_\alpha{:}A.\, f(x) = f(y) \Rightarrow x = y \qquad (6.2)$$

$$Surj \quad \equiv \quad \lambda f_{\alpha\beta} \textbf{.} \lambda A_{\alpha o} \textbf{.} \lambda B_{\beta o} \textbf{.} \forall x_\beta{:}B \textbf{.} \exists y_\alpha{:}A \textbf{.} f(y) = x \tag{6.3}$$

Note that in the definition of surjective we have to explicitly talk about elements of the codomain — and thus about the set they belong to — of the mapping f. On the contrary injective can be defined without specifying the actual set of the codomain.

Finally, we define the concepts of image and kernel of a mapping f, where the former is a subset of the codomain of f and the latter is a subset of the domain of f.

$$Im \quad \equiv \quad \lambda f_{\alpha\beta} \textbf{.} \lambda A_{\alpha o} \textbf{.} \lambda y_\beta \textbf{.} \exists x_\alpha{:}A \textbf{.} y = f(x) \tag{6.4}$$

$$Kern \quad \equiv \quad \lambda f_{\alpha\beta} \textbf{.} \lambda A_{\alpha o} \textbf{.} \lambda y_\beta \textbf{.} \lambda x_\alpha \textbf{.} [x \in A] \wedge [f(x) = y] \tag{6.5}$$

We verify that Im is indeed a subset of the codomain of f since $Im(f, A)_{\beta o}$ is a set of elements of type β. Likewise, $Kern$ is a set of elements of type α (i.e., a subset of the domain of f) if we have $Kern(f, A, y)$, where y is the trivial element of the codomain. For instance, if f is mapping between groups, y is the unit element of the target group.

6.3 Constructing Proofs

All homomorphism proofs are constructed with the `homomorphisms` strategy whose interactive variant we have already discussed in section 4.1.1. The strategy can employ 22 methods of which 9 are domain-specific. As an interactive strategy for Ω-ANTS it also contains 29 agents that test for the applicability of the single methods. To see this figure in the right perspective we have to take into account that from the 22 methods five are normalization or restriction methods and thus do not require any agents. As an automatic strategy for the *PPlanner* algorithm `homomorphisms` additionally contains 6 domain-specific control rules, which, for instance, ensure that if the homomorphism property has been applied in one direction it is not immediately applied in the opposite direction again, or that goals containing homomorphism statements involving meta-variables are preferred in order to constrain their instantiations as soon as possible.

The basic approach of the strategy to prove homomorphism problems is to first expand all definitions up to a point where the homomorphism property can be applied as often as possible; that is, if we have a homomorphism $f : A \to B$ we try to rewrite all occurrences of the operation on B into the homomorphic application of the operation on f. This transforms problems stated for B into equivalent problems on A, which are generally simpler to show. The proofs are then concluded by deriving the necessary properties from the definition of A.

The central method for homomorphism proofs is the *Homomorphism* method given in figure 6.1, which is basically a more complex version of the *HomOnDomain* method we have already discussed in chapter 4.1.1. Its task is to apply the homomorphism given in line L_1 backwards (here again the void justification indicates that L_1 can have an arbitrary justification). However, it is applied to a goal line containing an application of the operation $\diamond$ to elements actually belonging to the codomain of the homomorphism. This is given in line L_7 of the method by the schematic formula $\Phi[b_1 \diamond b_2]$, where Φ is an arbitrary proposition containing a subterm of the form $b_1 \diamond b_2$. When the method is applied the occurrence of $b_1 \diamond b_2$ in Φ is replaced by $f(mv_1 \circ mv_2)$ in line L_6, with $f(mv_i) = b_i$ for $i = 1, 2$ where the mv_i are meta-variables substituting for the actual elements of A. Additionally, we need to ensure that mv_1 and mv_2 are really elements of the domain A. Those

Method: *Homomorphism*	
Premises	$L_1, \oplus L_2, \oplus L_3, \oplus L_4, \oplus L_5, \oplus L_6$
Appl. Cond.	
Conclusions	$\ominus L_5$
Declarative Content	$(L_1)\ \Delta\ \vdash Hom(f,(A,\circ),(B,\diamond))$ $(L_2)\ \Delta\ \vdash mv_1 \in A$ $(Open)$ $(L_3)\ \Delta\ \vdash mv_2 \in A$ $(Open)$ $(L_4)\ \Delta\ \vdash f(mv_1) = b_1$ $(Open)$ $(L_5)\ \Delta\ \vdash f(mv_2) = b_2$ $(Open)$ $(L_6)\ \Delta\ \vdash \Phi[f(mv_1 \circ mv_2)]$ $(Open)$ $(L_7)\ \Delta\ \vdash \Phi[b_1 \diamond b_2]$ $(ApplyHomComplex\ L_1\ L_6\ L_2\ L_3\ L_4\ L_5)$

Figure 6.1: The *Homomorphism* method.

Method: *ElemOfDomain*	
Premises	$\ominus L_2, \oplus L_4$
Appl. Cond.	
Conclusions	$\ominus L_5$
Declarative Content	$(L_1)\ \Delta\ \vdash [c \in G] \wedge [f(c) = d]$ (Hyp) $(L_2)\ \Delta\ \vdash d \in Im(f,G)$ $(L_3)\ \Delta\ \vdash \exists x{:}G.\,[f(x) = d]$ $(\equiv_E\ L_2\ Im)$ $(L_4)\ \Delta \cup \{L_1\} \vdash \Phi$ $(Open)$ $(L_5)\ \Delta\ \vdash \Phi$ $(\exists_E Sort\ L_3\ L_4\ c)$

Figure 6.2: The *ElemOfDomain* method.

supplementary conditions are given in the extra four new open subgoals L_2 through L_5. The method does not have any additional application conditions. Similarly to the *HomOnDomain* method we have two agents $\mathfrak{S}^{\{L_1\}}_{\{\},\{\}}$ and $\mathfrak{G}^{\{L_7\}}_{\{L_1\},\{\}}$ to search for matching lines in a given partial proof.

Another important method to construct homomorphism proofs is the normalization method *ElemOfDomain* displayed in figure 6.2. It is applied as soon as a support line of the form given in L_2 of the method's declarative content occurs in the proof. It basically states if we have an element d in the image of a function f on a set G given, then there exists a $c \in G$ such that $f(c) = d$. The method introduces this fact as the new hypothesis L_1. This corresponds to a sorted elimination of the existential quantifier of the actual definition of image as given in the preceding section. This step is stated in L_3 of the declarative content but is only introduced into the proof when the method is expanded.

The effects of the method on the partial proof are that line L_1 is added as a new hypothesis for the considered subgoal. Thereby the old subgoal L_5 is deleted from the planning state and L_4, the line with the expanded hypotheses list, is added as new open subgoal. The support line L_2 is also deleted from the planning state. Although not explicitly specified, line L_1 is automatically added to the planning state as a new support line since it is a new hypothesis for the introduced subgoal. Since *ElemOfDomain* is a normalization method it is automatically and exhaustively applied if possible after each interaction step. Like all normalization methods it does not have any agents on its own to test its applicability.

The second domain-specific normalization method of the `homomorphisms` strat-

L_2.	L_2	$\vdash Group(G,\circ)$	(Hyp)
L_3.	L_3	$\vdash Group(H,\star)$	(Hyp)
L_4.	L_4	$\vdash Hom(f,(G,\circ),(H,\star))$	(Hyp)
L_{23}.	$\mathcal{H}_1$	$\vdash f(mv_4) = f(GroupUnit(G,\circ))$	$(=Reflexivity)$
L_{22}.	$\mathcal{H}_1$	$\vdash mv_4 \in G_{\{mv_4 \leftarrow GroupUnit(G,\circ)\}}$	$(UnitInGroup\ L_2)$
L_{21}.	$\mathcal{H}_1$	$\vdash \exists y{:}G\blacksquare f(GroupUnit(G,\circ)) = f(y)$	$(\exists_I Sort\ L_{22}\ L_{23})$
L_5.	$\mathcal{H}_1$	$\vdash mv_1 \in Im(f,G)$	$(\equiv_I\ L_{21}\ Im)$
L_8.	L_8	$\vdash c \in Im(f,G)$	(Hyp)
L_{10}.	L_{10}	$\vdash [d \in G] \land [f(d) = c]$	(Hyp)
L_{12}.	L_{10}	$\vdash d \in G$	$(\land_E\ L_{10})$
L_{13}.	L_{10}	$\vdash f(d) = c$	$(\land_E\ L_{10})$
L_{14}.	$\mathcal{H}_3$	$\vdash mv_2 \in G$	$(UnitInGroup\ L_2)$
L_{20}.	$\mathcal{H}_3$	$\vdash d \in G$	$(Weaken\ L_{12})$
L_{19}.	$\mathcal{H}_3$	$\vdash mv_2 \circ d = d_{\{mv_2 \leftarrow GroupUnit(G,\circ)\}}$	$(GroupUnit\ L_2\ L_{20})$
L_{18}.	$\mathcal{H}_3$	$\vdash f(mv_2 \circ mv_3) = f(mv_3)$	$(EqualFunc\ L_{19})$
L_{17}.	$\mathcal{H}_3$	$\vdash f(mv_3) = c$	$(Weaken\ L_{13})$
L_{16}.	$\mathcal{H}_3$	$\vdash f(mv_2) = mv_{1\{mv_1 \leftarrow f(mv_2)\}}$	$(=Reflexivity)$
L_{15}.	$\mathcal{H}_3$	$\vdash mv_3 \in G_{\{mv_3 \leftarrow d\}}$	$(Weaken\ L_{12})$
L_{11}.	$\mathcal{H}_3$	$\vdash mv_1 \star c = c$	$(Homomorphism$ $L_4\ L_{18}\ L_{14}\ L_{15}\ L_{16}\ L_{17})$
L_9.	$\mathcal{H}_2$	$\vdash mv_1 \star c = c$	$(ElemOfDomain\ L_8\ L_{11})$
L_7.	$\mathcal{H}_1$	$\vdash \forall x{:}Im(f,G)\blacksquare [mv_1 \star x = x]$	$(\forall_I Sort\ L_9)$
L_6.	$\mathcal{H}_1$	$\vdash LeftUnit(Im(f,G),\star,mv_1)$	$(\equiv_I\ L_7\ LeftUnit)$
L_1.	$\mathcal{H}_1$	$\vdash \exists e{:}Im(f,G)\blacksquare LeftUnit(Im(f,G),\star,e)$	$(\exists_I Sort\ L_5\ L_6)$

$\mathcal{H}_1 = \{L_2, L_3, L_4\}, \quad \mathcal{H}_2 = \{L_2, L_3, L_4, L_8\}, \quad \mathcal{H}_3 = \{L_2, L_3, L_4, L_8, L_{10}\}$

Table 6.2: Proof of a homomorphism theorem.

egy is the *ElemOfKernel* method. It performs essentially the same task as the *ElemOfDomain* method, albeit for elements of the kernel of a homomorphism.

6.4 Example

As an example for the proofs of homomorphism theorems consider the problem

$$[Group(G,\circ) \land Group(H,\star) \land Hom(f,(G,\circ),(H,\star))]$$
$$\Rightarrow [\exists e{:}Im(f,G)\blacksquare LeftUnit(Im(f,G),\star,e)],$$

which is a lemma for the proof of theorem 1 in table 6.1 when we use the definition of a group as given in 5.1 with the refinement of axiom $\mathcal{G}3^*$, the existence of a left unit element.

The proof of this lemma is given in table 6.2. Here we assume that the implication has already been split and we derive the succedent given in line L_1 from the three assumptions given as hypotheses in lines L_2 to L_4. To ameliorate readability some of the hypotheses lists have been abbreviated.

The first step in the proof is an interactive application of the $\exists_I Sort$ method, which leads to the two new subgoals L_5 and L_6. The latter subgoal states that mv_1 is actually a left unit in the image of f on G, whereas the former subgoal requires us to show that the term that will eventually be substituted for mv_1 is actually an element of that image. We concentrate first on the proof of the subgoal in line L_6.

The next two proof steps are again interactive: First the expansion of the definition of $LeftUnit$ and subsequently the elimination of the universal quantifier in line L_7. Eliminating the sorted quantifier leads also to the introduction of the new hypothesis L_8 into the proof. This, in turn, triggers the first automatic application of a normalization method, namely $ElemOfDomain$. It decomposes the statement that c is an element of the image of f in line L_8 into the conjunction of line L_{10} and introduces also the new subgoal L_{11}. Line L_{10} is even further decomposed by another normalization method, $\wedge_E$, which is again applied automatically.

At this point of the proof all possible quantifiers are eliminated and the definitions are expanded. Hence, Ω-Ants's next suggestion is the application of the *Homomorphism* method to line L_{11}. This leads to the introduction of five new open subgoals L_{14} to L_{18}. The three most tedious of those subgoals, L_{15} through L_{17}, are closed immediately and automatically by restriction methods. Thereby two of the meta-variables, mv_2 and mv_3, are matched and substituted.

However, this substitution is not carried out directly during the proof planning process but merely added as a constraint on the meta-variable. Nevertheless, from there on the proof planner treats any occurrence of the meta-variables as if they were substituted. Once a complete proof plan is constructed all computed substitutions for meta-variables are applied. This treatment of meta-variable substitutions simplifies the backtracking procedure of the proof planner. In our examples we examine the proof planning process in progress. Hence, we introduce first the meta-variables themselves (i.e., to demonstrate when meta-variables arise). Once the meta-variable is constrained we indicate this with a subscript to a formula and will use the substituted term from there on in the proof. For example, in table 6.2 the substitution for the meta-variable mv_3 is introduced by the application of the $Weaken$ method matching the term $mv_3 \in G$ against $d \in G$ in line L_{12}. Thus, in all lines that are introduced after the application of $Weaken$ to L_{15} we write d instead of mv_3. On the contrary line L_{18} still contains mv_3 since it was introduced into the proof before the meta-variable was bound.

L_{18} is also the subgoal we have to consider next. We first decompose it by discarding the topmost function symbols on both sides of the equality since they are both f. This leaves us to show the equation $mv_2 \circ d = d$ holds. Because $\circ$ is the operation of G, which is a group according to line L_2, the *GroupUnit* method is applicable. It then remains to show that d is actually an element of G which is automatically closed by a $Weaken$ application. Additionally, the *GroupUnit* method binds the meta-variable mv_2 to the unit element of the group G indicated by $\{mv_2 \leftarrow \mathit{GroupUnit}(G, \circ)\}$. The expression $\mathit{GroupUnit}(G, \circ)$ is a way to actually refer to the unit element of G in the proof similar to the *StructUnit* predicate given in chapter 5.3.2. While in the pen and paper proof in mathematics we would simply use a meaningful name such as e_G for the constant denoting the unit element of G this is not possible in the ND proof, since constants cannot have any intrinsic semantical meaning. Thus, introducing $\mathit{GroupUnit}(G, \circ)$ instead of a simple constant enables us to actually denote the distinct element of G, which is the unit element with respect to $\circ$. The knowledge how and when to introduce such a distinct element into a proof planning is given in certain domain-specific methods. In our example the *GroupUnit* identifies the given equation as part of the axiom for the unit element and hence equates mv_2 with $\mathit{GroupUnit}(G, \circ)$.

With mv_2 representing the unit element of G we can now also close line L_{14}, the last subgoal produced by the application of the *Homomorphism* method. It essentially states that the unit element is actually an element of the group, which can be justified with the *UnitInGroup* method.

Having closed this branch of the proof the planner turns back to line L_5, the second of the subgoals produced by the very first method application. Considering the constraints for the meta-variable mv_1 computed so far the formula actually reads: $f(\mathit{GroupUnit}(G,\circ)) \in \mathit{Im}(f,G)$. After expanding the definition of image and subsequently eliminating the existential quantifier in line L_{21} there are two subgoals left to show, the lines L_{22} and L_{23}. The latter is closed automatically by an application of the restriction method $=\mathit{Reflexivity}$, which also binds the meta-variable mv_4 to $\mathit{GroupUnit}(G,\circ)$. This constraint is given as subscript of the formula of line L_{22} in order to preserve the readability of line L_{23}. With mv_4 being the unit element of G the final subgoal can be justified interactively with another application of the *UnitInGroup* method.

Although the example proof is relatively long it has been achieved with exactly ten interaction steps. In particular the user does not have to be concerned with executing the intermediate normalization steps since both normalization and restriction methods are always performed automatically and exhaustively without interaction. As the example demonstrates this helps to focus the user on the actually interesting steps in the proof. While this aids to fruitfully employ interactive proof planning in the context of tutor systems, the interaction steps are by no means necessary. In fact, all proofs presented in this chapter can be planned fully automatic.

6.5 Summary of Chapter 6

This chapter showed the use of Ω-ANTS as an algorithm in MULTI to support interactive proof planning. We applied this mechanism to 28 homomorphism theorems, which can be either solved interactively with the Ω-ANTS algorithm or planned automatically. We have a choice of regular planning methods with agents that have to be applied interactively together with a set of normalization and restriction methods, which are applied automatically after each proof step. This enables us to construct proofs with a relatively small number of interaction steps and thus focusing the user on the important and interesting steps in the proof.

Chapter 7

Exploring Properties of Residue Classes

This chapter presents a case study on the combination of multi-strategy proof planning both with computer algebra as given in chapter 4.2 and with Ω-ANTS as described in chapter 4.1.2. The domain of the case study is the exploration of simple algebraic properties of sets of residue classes over the integers. The proofs in this domain can be carried out with different proof techniques, which are modeled as strategies of MULTI. In particular we have three different strategies, each one exemplifying one of the following points:

1. The use of computer algebra systems in control rules to guide the proof planning process.
2. The application of symbolic computation within the method applications.
3. The exploitation of Ω-ANTS search mechanism to find applicable theorems during method matching.

To gain and prove a fair number of examples we have conceived a way to automatically classify large numbers of residue class sets together with binary operations in terms of the algebraic structure they form. This enables to conduct large numbers of experiments in order to test the robustness and the usefulness of the implemented strategies. Moreover, the classification process demonstrates another way of fruitfully employing computer algebra within an environment such as ΩMEGA. The case study has been first reported in [153].

We shall first introduce the problem domain and the necessary formalizations. Then we present the three different proof techniques, their respective implementation in strategies of MULTI and an example for the constructed proofs. We shall then describe the automatic classification process that enabled us to perform a large number of experiments with the presented techniques. An account and a discussion of these experiments wrap up this chapter.

7.1 Problem Domain

The aim of the case study is to explore simple algebraic properties of residue class sets over the integers together with given binary operations. In our terminology a *residue class set* over the integers is either the set of all congruence classes modulo

1.	(a) $Closed(RS_n, \circ)$	(b) $\neg Closed(RS_n, \circ)$
2.	(a) $Assoc(RS_n, \circ)$	(b) $\neg Assoc(RS_n, \circ)$
3.	(a) $\exists e{:}{\scriptstyle RS_n}\mathbf{.}Unit(RS_n, \circ, e)$	(b) $\neg\exists e{:}{\scriptstyle RS_n}\mathbf{.}Unit(RS_n, \circ, e)$
4.	(a) $Inverse(RS_n, \circ, e)$	(b) $\neg Inverse(RS_n, \circ, e)$
5.	(a) $Divisors(RS_n, \circ)$	(b) $\neg Divisors(RS_n, \circ)$
6.	(a) $Commu(RS_n, \circ)$	(b) $\neg Commu(RS_n, \circ)$
7.	(a) $Distrib(RS_n, \circ, \star)$	(b) $\neg Distrib(RS_n, \circ, \star)$

Table 7.1: Theorems for properties of residue class structures.

an integer n (i.e., $\mathbb{Z}_n$) or an arbitrary subset of $\mathbb{Z}_n$. Concretely, we are dealing with sets of the form $\mathbb{Z}_3, \mathbb{Z}_5, \mathbb{Z}_3\backslash\{\bar{1}_3\}, \mathbb{Z}_5\backslash\{\bar{0}_5\}$, $\{\bar{1}_6, \bar{3}_6, \bar{5}_6\}, \ldots$ where $\bar{1}_3$ denotes the congruence class 1 modulo 3. If c is an integer we write also $cl_n(c)$ for the congruence class c modulo n. Additionally we allow for direct products of residue class sets of arbitrary yet finite length; thus, we can have sets of the form $\mathbb{Z}_3 \otimes \mathbb{Z}_5$, $\mathbb{Z}_3\backslash\{\bar{1}_3\} \otimes \mathbb{Z}_5\backslash\{\bar{0}_5\} \otimes \{\bar{1}_6, \bar{3}_6, \bar{5}_6\}$, etc.

A ***binary operation*** $\circ$ on a residue class set is given in λ-function notation. $\circ$ can be of the form $\lambda x\mathbf{.}\lambda y\mathbf{.}x$, $\lambda x\mathbf{.}\lambda y\mathbf{.}y$, $\lambda x\mathbf{.}\lambda y\mathbf{.}c$ where c is a constant congruence class (e.g., $\bar{1}_3$), $\lambda x\mathbf{.}\lambda y\mathbf{.}x\bar{+}y$, $\lambda x\mathbf{.}\lambda y\mathbf{.}x\bar{*}y$, $\lambda x\mathbf{.}\lambda y\mathbf{.}x\bar{-}y$, where $\bar{+}$, $\bar{*}$, $\bar{-}$ denote addition, multiplication, and subtraction on congruence classes over the integers, respectively. Furthermore, $\circ$ can be any combination of the basic operations with respect to a common modulo factor, for example, $\lambda x\mathbf{.}\lambda y\mathbf{.}(x\bar{+}\bar{1}_3)\bar{-}(y\bar{+}\bar{2}_3)$. For direct products of residue class sets the operation is a combination of the single binary operations for the emerging element tuples, for example, $\lambda x\mathbf{.}\lambda y\mathbf{.}x\bar{+}y \oplus \lambda x\mathbf{.}\lambda y\mathbf{.}x\bar{*}y$, etc.

We shall consider residue class sets RS_n modulo n with either one binary operation $\circ$ or two binary operations $\circ$ and $\star$. Both $\circ$ and $\star$ are required to be operations with respect to the modulo factor n of the residue class. We call such combinations ***residue class structures*** (or simply ***structures***) and denote them by $(RS_n, \circ)$ or $(RS_n, \circ, \star)$, respectively.

The theorems we prove for properties of residue class structures are given in table 7.1. The theorems on the lefthand side of the table all contain the assertion that a certain property holds for a residue class structure. In contrast the theorems on the righthand side contain the respective negated assertions, that some property does not hold. For example, theorem 1a states that the residue class set RS_n is closed under the operation $\circ$, whereas its negation, theorem 1b, states that RS_n is not closed with respect to $\circ$.

For structures with one operation, $(RS_n, \circ)$, we are essentially interested in the group properties given in the equations (5.2) to (5.5) and (5.8) in chapter 5.3.1, which corresponds to the theorems under 1 to 5. In addition we also consider whether a residue class structure is commutative, given in line 6. And, given a structure $(RS_n, \circ, \star)$ consisting of a residue class set together with two binary operations, we are interested in the distributivity of the two operations, which corresponds to the theorems 7a and 7b.

7.2 Formalization

Since the algebraic properties of the theorems in table 7.1 were already introduced in chapter 5, we have to formalize only the notion of a residue class set and some related concepts. Thus, it suffices to define the notion of residue class structures by giving here the relevant notions and theorems for residue class sets and operations on them. Furthermore, we introduce the necessary definitions for direct products of residue classes. The proofs of the theorems given in this section are interactively constructed in ΩMEGA but are not explicitly discussed here.

7.2.1 Residue Class Sets

We start with the notion of a congruence class to gain the basic elements we need to construct residue class sets

$$cl \equiv \lambda n_\nu \mathbf{.} \lambda m_\nu \mathbf{.} \lambda x_\nu \mathbf{.} [\mathbb{Z}(x)] \wedge [(x \; mod \; n) = m] \tag{7.1}$$

Provided cl is applied to two arguments n and m, we have the set containing all integers x such $(x \; mod \; n) = m$. One crucial point of the definition is that the value for n can range over all numbers. However, the application of mod ensures that the above expression is only meaningful if n is an integer, which is in particular not zero. In the following we generally write a congruence class m modulo n as $cl_n(m)$.

Having congruence classes as building blocks available we can define residue class sets as

$$RS \equiv \lambda n_\nu \mathbf{.} \lambda r_{\nu o} \mathbf{.} \exists m_\nu {:} \mathbb{N} \mathbf{.} [r = cl_n(m)] \wedge [NonEmpty(cl_n(m))] \tag{7.2}$$

Here $NonEmpty$ corresponds to predicate defined in chapter 5.3 equation (5.1). We write a residue class set over an integer n as RS_n. We can now show three theorems for residue class sets.

$$\forall n{:}\mathbb{Z}\mathbf{.} NonEmpty(RS_n) \Leftrightarrow [n \neq 0] \tag{7.3}$$

$$\forall n{:}\mathbb{Z}\mathbf{.} \forall m{:}\mathbb{N}\mathbf{.} [cl_n(m) \in RS_n] \Leftrightarrow [m < |n|] \tag{7.4}$$

$$\forall n{:}\mathbb{Z}\mathbf{.} \forall m{:}\mathbb{N}\mathbf{.} NonEmpty(cl_n(m)) \Rightarrow [m \in cl_n(m)] \tag{7.5}$$

Theorem (7.3) means that residue class sets are nonempty iff the modulo factor is distinct from 0. The theorems (7.4) and (7.5) state respectively that a congruence class is included in a residue class set iff its representative is smaller than the absolute value of the modulo factor and that each nonempty congruence class contains at least its representative.

We then need a means to access the representative of a congruence class, therefore we define a predicate Res that gives us the residue m of all elements of a congruence class $cl_n(m)$.

$$Res \equiv \lambda c_{\nu o} \mathbf{.} \lambda n_\nu \mathbf{.} \iota m_\nu \mathbf{.} \forall x_\nu \mathbf{.} [x \in c] \Rightarrow [x \; mod \; n = m] \tag{7.6}$$

Res takes two arguments; the first is the congruence class and the second is the corresponding modulo factor. Here ι is the description operator defined in chapter 2.1.1, definition 2.4, which acts like an additional quantifier. Its meaning here is intuitively that m represents *that* uniquely determined positive integer m that is the residue modulo n of all elements of the congruence class c.

We can again prove that the predicate Res is well defined and additionally that it has indeed the desired properties.

$$\forall n_\nu{:}\mathbb{Z}\mathbf{.} \forall c_{\nu o}{:}RS_n\mathbf{.} \exists x_\nu{:}\mathbb{N}\mathbf{.} [Res(c,n) = x] \;\wedge\; [x \in c] \tag{7.7}$$

$$\forall n_\nu{:}\mathbb{Z}\mathbf{.} \exists m_\nu{:}\mathbb{N}\mathbf{.} [[n \neq 0] \wedge [m < |n|]] \Rightarrow [Res(cl_n(m),n) = m] \tag{7.8}$$

$$\forall n_\nu{:}\mathbb{Z}\mathbf{.} \forall c_{\nu o}{:}RS_n\mathbf{.} \forall x_\nu\mathbf{.} [Res(c,n) = x] \Rightarrow [x \in c] \wedge [x \in \mathbb{N}] \tag{7.9}$$

$$\forall n_\nu{:}\mathbb{Z}\mathbf{.} \forall c_{\nu o}{:}RS_n\mathbf{.} \forall d_{\nu o}{:}RS_n\mathbf{.} [c = d] \Leftrightarrow [Res(c,n) = Res(d,n)] \tag{7.10}$$

The last theorem gives us a means to transform statements on congruence classes into corresponding statements on integers modulo a factor n. This theorem is used in the proofs of theorems over operations on residue classes, which we shall see in the sequel.

7.2.2 Operations on Congruence Classes

We first define the basic operations on congruence classes we shall consider, namely addition, multiplication, and subtraction.

$$\bar{+} \equiv \lambda r_{\nu o}\centerdot\lambda s_{\nu o}\centerdot\lambda z_{\nu}\centerdot\exists x_{\nu}{:}r\centerdot\exists y_{\nu}{:}s\centerdot z = x + y \tag{7.11}$$

$$\bar{*} \equiv \lambda r_{\nu o}\centerdot\lambda s_{\nu o}\centerdot\lambda z_{\nu}\centerdot\exists x_{\nu}{:}r\centerdot\exists y_{\nu}{:}s\centerdot z = x * y \tag{7.12}$$

$$\bar{-} \equiv \lambda r_{\nu o}\centerdot\lambda s_{\nu o}\centerdot\lambda z_{\nu}\centerdot\exists x_{\nu}{:}r\centerdot\exists y_{\nu}{:}s\centerdot z = x - y \tag{7.13}$$

We can observe that the definitions (7.11) to (7.13) make no restriction on the congruence classes involved; that is, they do not have to be necessarily congruence classes with respect to the same modulo factor. However, in practice operations between congruence classes with differing modulo factor are of little use and for the following theorems it is therefore ensured that the arguments of the respective operations are from the same residue class set.

$$\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot (c \mathbin{\bar{+}} d) \in RS_n \tag{7.14}$$

$$\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot (c \mathbin{\bar{*}} d) \in RS_n \tag{7.15}$$

$$\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot (c \mathbin{\bar{-}} d) \in RS_n \tag{7.16}$$

$$\begin{aligned}&\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot\forall p_{\nu}\centerdot\forall q_{\nu}\centerdot\\ &\quad [[Res(c,n) = p] \wedge [Res(d,n) = q]] \Rightarrow [Res((c \mathbin{\bar{+}} d),n) = (p+q)\ mod\ n]\end{aligned} \tag{7.17}$$

$$\begin{aligned}&\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot\forall p_{\nu}\centerdot\forall q_{\nu}\centerdot\\ &\quad [[Res(c,n) = p] \wedge [Res(d,n) = q]] \Rightarrow [Res((c \mathbin{\bar{*}} d),n) = (p*q)\ mod\ n]\end{aligned} \tag{7.18}$$

$$\begin{aligned}&\forall n{:}\mathbb{Z}\centerdot\forall c{:}RS_n\centerdot\forall d{:}RS_n\centerdot\forall p_{\nu}\centerdot\forall q_{\nu}\centerdot\\ &\quad [[Res(c,n) = p] \wedge [Res(d,n) = q]] \Rightarrow [Res((c \mathbin{\bar{-}} d),n) = (p-q)\ mod\ n]\end{aligned} \tag{7.19}$$

The first three theorems simply state that a residue class set is a closed set with respect to our three operations on congruence classes. The theorems (7.17) to (7.19) provide a way to shift from an operation on congruence classes to the corresponding operation on integers. To prove these theorems we need to apply the theorem (7.10).

7.2.3 Direct Products

We define direct products of residue classes via iterated pairing of arbitrary sets. For this we first need to define the notion of pairs of single elements with the following pairing function:

$$Pair \equiv \lambda x_{\alpha}\centerdot\lambda y_{\beta}\centerdot\lambda g_{\alpha\beta o}\centerdot g(x,y) \tag{7.20}$$

In order to access the single elements of a pair we need to define two projections for the left and the right element of the pair, respectively. The definitions of the projections and the pairing functions are identical with those given in ANDREW's book [7] on page 185.

$$LProj \equiv \lambda p_{(\alpha\beta o)o}\centerdot\iota x_{\alpha}\centerdot\exists y_{\beta}\centerdot p = Pair(x,y) \tag{7.21}$$

$$RProj \equiv \lambda p_{(\alpha\beta o)o}\centerdot\iota y_{\beta}\centerdot\exists x_{\alpha}\centerdot p = Pair(x,y) \tag{7.22}$$

Note that the definitions of pair and the two projection functions are basically the same as the definitions of the construction operator for lists # and its access functions in chapter 3.6.1. However, we forgo reusing these functions in order to distinguish pairs from lists, although Cartesian products of more than two sets are recursively composed pairs and thus essentially similar to lists.

The predicate *Pair* and its projection functions enable us to prove a series of simple, nevertheless useful theorems.

$$\forall x_\alpha\text{.}\forall y_\beta\text{.}\ \ LProj(Pair(x,y)) = x \tag{7.23}$$

$$\forall x_\alpha\text{.}\forall y_\beta\text{.}\ \ RProj(Pair(x,y)) = y \tag{7.24}$$

$$\forall x_\alpha\text{.}\forall y_\beta\text{.}\forall a_\alpha\text{.}\forall b_\beta\text{.}\ \ [Pair(x,y) = Pair(a,b)] \Leftrightarrow [x = a \wedge y = b] \tag{7.25}$$

Theorem (7.23) and (7.24) state that the projections indeed return the left and right element of the pair, respectively, and theorem (7.25) is an aid to deal with equalities between pairs.

We can now define the direct product of two sets as the set of all pairs of elements of the respective sets; that is,

$$\otimes \equiv \lambda U_{\alpha o}\text{.}\lambda V_{\beta o}\text{.}\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\text{.}[LProj(p) \in U] \wedge [RProj(p) \in V] \tag{7.26}$$

Finally, we define operations on direct products as pairs of the operations of the single sets of the direct product.

$$\begin{aligned}\times \equiv \lambda U_{\alpha o}\text{.}\lambda V_{\beta o}\text{.}\lambda \circ^1_{\alpha\alpha\alpha}\text{.}\lambda \circ^2_{\beta\beta\beta}\text{.}\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\text{.}\lambda q_{(\alpha\beta)((\alpha\beta o)o)}\text{.}\\ Pair(LProj(p) \circ^1 LProj(q), RProj(p) \circ^2 RProj(q))\end{aligned} \tag{7.27}$$

In this thesis we write pairs of operations as $(\circ^1 \times \circ^2)$. Moreover, we write direct products of more than two sets as $U_1 \otimes U_2 \otimes \ldots \otimes U_n$, thus omitting the brackets. Likewise, we omit the brackets for the operations on such sets.

7.3 Planning Proofs of Simple Properties

In order to automatically analyze the properties of residue class sets we use the multi-strategy proof planner MULTI. MULTI can use different strategies to construct an appropriate proof plan where each strategy implements a different proof technique. However, not all proof techniques can be applied to all possibly occurring problems. Thus, the advantage of using MULTI is that fast but not always successful strategies can be tested first, and if they fail slower but more reliable strategies can be employed. Moreover, strategies can be intermixed in the sense that certain subgoals arising during the application of one strategy can be proved with a different technique.

In particular we have three strategies implemented for proving single properties of residue class structures, namely

1. exhaustive case analysis,
2. equational reasoning, and
3. application of already known theorems.

The proof planner tries to apply the strategies in the order 3 to 1 since strategy 3 is generally the fastest to solve a problem and strategy 1 is the most reliable of the three strategies.

In the sequel we elaborate each strategy using examples for the type of proofs they produce. We shall point out the major differences while trying to avoid the tedious details and mention advantages and weaknesses of each strategy as we go along. Furthermore, we emphasize where the combinations of proof planning with computer algebra and Ω-ANTS come into play and how they are realized in the different strategies.

L_1.	L_1	$\vdash cl_2(c) \in \mathbb{Z}_2$	(Hyp)
L_2.	L_1	$\vdash c \in \{0,1\}$	$(ConResclSet\ L_1)$
L_3.	L_3	$\vdash c = 0$	(Hyp)
		$\vdots$	
L_{12}.	L_1, L_3	$\vdash \exists y{:}\mathbb{Z}_2. [cl_2(c)\bar{+}y{=}\bar{0}_2] \wedge [y\bar{+}cl_2(c){=}\bar{0}_2]$	$(\exists_I Resclass\ L_{11}\ L_{10})$
L_{13}.	L_{13}	$\vdash c = 1$	(Hyp)
L_{14}.	L_1, L_{13}	$\vdash 0 = 0$	$(=Reflexivity)$
L_{15}.	L_1, L_{13}	$\vdash mv \in \{0,1\}$	$(\vee_{Ir}\ L_{14})$
L_{16}.	L_1, L_{13}	$\vdash 0 = 0$	$(=Reflexivity)$
L_{17}.	L_1, L_{13}	$\vdash 0 = 0$	$(=Reflexivity)$
L_{18}.	L_1, L_{13}	$\vdash (1+c)\ mod\ 2 = 0\ mod\ 2$	$(SimplNum\ L_{13}\ L_{16})$
L_{19}.	L_1, L_{13}	$\vdash (c+1)\ mod\ 2 = 0\ mod\ 2$	$(SimplNum\ L_{13}\ L_{17})$
L_{20}.	L_1, L_{13}	$\vdash [(c{+}1)\ mod\ 2 = 0\ mod\ 2] \wedge [(1{+}c)\ mod\ 2 = 0\ mod\ 2]$	$(\wedge_I\ L_{18}\ L_{19})$
L_{21}.	L_1, L_{13}	$\vdash [cl_2(c)\bar{+}cl_2(mv) = \bar{0}_2] \wedge [cl_2(mv)\bar{+}cl_2(c) = \bar{0}_2]_{\{mv \leftarrow 1\}}$	$(ConCongCl\ L_{20})$
L_{22}.	L_1, L_{13}	$\vdash \exists y{:}\mathbb{Z}_2. [cl_2(c)\bar{+}y = \bar{0}_2] \wedge [y\bar{+}cl_2(c) = \bar{0}_2]$	$(\exists_I Resclass\ L_{21}\ L_{15})$
L_{23}.	L_1	$\vdash \exists y{:}\mathbb{Z}_2. [cl_2(c)\bar{+}y = \bar{0}_2] \wedge [y\bar{+}cl_2(c) = \bar{0}_2]$	$(\vee_E^{**}\ L_2\ L_{12}\ L_{22})$
L_{24}.		$\vdash \forall x{:}\mathbb{Z}_2. \exists y{:}\mathbb{Z}_2. [x\bar{+}y = \bar{0}_2] \wedge [y\bar{+}x = \bar{0}_2]$	$(\forall_I Resclass\ L_{23})$
L_{25}.		$\vdash Inverse(\mathbb{Z}_2, \lambda x. \lambda y. x\bar{+}y, \bar{0}_2)$	$(\equiv_I\ L_{24}\ Inverse)$

Table 7.2: Proof with the `TryAndError` strategy.

7.3.1 Exhaustive Case Analysis

The motivation for the first strategy, called `TryAndError`, is to implement a reliable approach of proving a property of a residue class set. It proceeds by rewriting statements on residue classes into corresponding statements on integers, especially by transforming the residue class set into a set of corresponding integers. It then exhaustively checks all possible combinations of these integers with respect to the property we have to prove or to refute. This approach is possible since in our problems the quantified variables range always over finite domains. `TryAndError` proceeds in two different ways, depending on whether (1) a universally or (2) an existentially quantified formula has to be proved. Both cases can be observed in the example proof of the theorem that $\mathbb{Z}_2$ has inverses with respect to the operation $\lambda x. \lambda y. x\bar{+}y$ and the unit element $\bar{0}_2$, given in table 7.2.

In case (1) a split over all the elements in the set involved is performed and the property is proved for every single element separately. We observe this in the proof of the universally quantified formula in line L_{24}. An application of the method $\forall_I Resclass$ to L_{24} yields the lines L_{23}, L_1, and L_2. $\forall_I Resclass$ is a method dual to $\exists_I Resclass$ that has been explained in section 2.2.4.1. The disjunction contained in L_2 ($c \in \{0,1\}$ can be viewed as $c = 0 \vee c = 1$) triggers the first case split with the application of $\vee_E^{**}$. Subsequently MULTI tries to prove the goal in line L_{23} twice (in the lines L_{12} and L_{22}), once assuming $c = 0$ (in line L_3) and once assuming $c = 1$ (in line L_{13}).

In case (2) the single elements of the set involved are examined until one is found for which the property in question holds. In our example this is, for instance, done after the application of the method $\exists_I Resclass$ to L_{22} yielding the lines L_{15} and L_{21}. The case analysis is then performed by successively choosing different possible values for mv with the $\vee_{Ir}$ and $\vee_{Il}$ methods that split disjunctive goals into the left or right disjunct, respectively. In our example mv is either 0 or 1 as given in line L_{15}. For a selected instantiation MULTI can then either finish the proof or — if the proving attempt fails — it backtracks to test the next instantiation. To minimize this search for a suitable instantiation of a meta-variable the `TryAndError` strategy enables MULTI to invoke the `select-instance` control rule on occurring meta-variables.

select-instance is a control rule of the type described in chapter 4.2. It is called directly after the introduction of a meta-variable mv and provides the information how to instantiate this meta-variable as a substitution $\{mv \leftarrow t\}$. But, as already explained in chapter 6.4 this substitution is not carried out directly during the proof planning process but merely added as a constraint on the meta-variable. So, for example, in table 7.2 we introduce the suitable instantiation 1 for the meta-variable mv in line L_{20} and indicate this by the substitution $\{mv \leftarrow 1\}$ in line L_{21}. To compute hints that simplify the proof select-instance employs computer algebra systems. Depending on what type of hint is required, it either uses ΩMEGA internal routines, GAP, or MAPLE. In detail, we have a function that constructs multiplication table for a given structure. This is used to check for closure and the existence of divisors and compute appropriate hints. If the multiplication table is closed, it can also be passed to GAP in order to check whether the given structure is associative, contains a unit element and inverses. According to GAP's results there are then hints constructed by the control rule.

For instance, if GAP can compute a unit element for a given semi-group this element is returned. In case GAP fails to find a unit element the multiplication table that has been constructed in ΩMEGA is used to determine the set of elements that suffice to refute the existence of a unit element. A special case is the failure of the query for associativity and commutativity, since then we try to use MAPLE to compute a particular solution for the respective equation. If such a non-general solution exists it is exploited to determine a tuple of elements for which the property in question (i.e., associativity or commutativity) does not hold.

In our example select-instance is called to compute the inverse of $\bar{1}_2$ in $\mathbb{Z}_2$, which is again $\bar{1}_2$. The corresponding integer 1 of this result is used as instantiation of the meta-variable mv throughout the rest of the proof (i.e., from line L_{20} on backwards).

After eliminating all quantifiers and performing all possible case splits the strategy reduces all remaining statements on residue and congruence classes to statements on integers using the *ConCongCl* method. This method employs essentially the theorems (7.17), (7.18), and (7.19) to convert expressions involving congruence classes to the corresponding expressions involving integers. The resulting statements are then solved by numerical simplification and basic equational reasoning.

The TryAndError strategy is designed to be applicable to every type of problem in our problem domain and ideally should also be always able to solve them[1]. However, the strategy has the major disadvantage that it has to wade painstakingly through all possible cases. This leads especially for large residue class sets to lengthy proofs, which can take quite long for the planner to construct.

7.3.2 Equational Reasoning

The aim of the second strategy, called EquSolve, is to use as much as possible equational reasoning to prove properties of residue classes. Similarly to the TryAndError strategy it converts statements on residue classes into corresponding statements on integers. But instead of then checking the validity of the statements for all possible cases, it tries to solve occurring equations in a general way. We observe this approach

[1] Since we cannot yet formally prove completeness for a strategy we have to rely on experiments to justify this statement. In our conducted experiments it turned out that the strategy can indeed solve all smaller problems, but that an exhaustive case analysis is no longer feasible for large problems (see section 7.5).

L_1.	$L_1 \vdash c'_1 \in \mathbb{Z}_2$	(Hyp)
L_2.	$L_1 \vdash c \in \{0,1\}$	$(ConResclSet\ L_1)$
L_{15}.	$L_1 \vdash mv \in \{0,1\}$	$(Weaken\ L_2)$
L_{18}.	$L_1 \vdash (mv + c)\ mod\ 2 = 0\ mod\ 2_{\{mv \leftarrow c\}}$	$(SolveEqu)$
L_{19}.	$L_1 \vdash (c + mv)\ mod\ 2 = 0\ mod\ 2$	$(SolveEqu)$
L_{20}.	$L_1 \vdash [(c + mv)\ mod\ 2 = 0\ mod\ 2] \wedge [(mv + c)\ mod\ 2{=}0\ mod\ 2]$	$(\wedge_I\ L_{19}\ L_{18})$
L_{21}.	$L_1 \vdash [cl_2(c)\bar{+}cl_2(mv){=}\bar{0}_2] \wedge [cl_2(mv)\bar{+}cl_2(c){=}\bar{0}_2]$	$(ConCongCl\ L_{20})$
L_{22}.	$L_1 \vdash \exists y{:}\mathbb{Z}_2{\scriptscriptstyle\blacksquare} [cl_2(c)\bar{+}y = \bar{0}_2] \wedge [y\bar{+}cl_2(c) = \bar{0}_2]$	$(\exists_I Resclass\, L_{21} L_{15})$
L_{24}.	$\vdash \forall x{:}\mathbb{Z}_2{\scriptscriptstyle\blacksquare} \exists y{:}\mathbb{Z}_2{\scriptscriptstyle\blacksquare} [x\bar{+}y = \bar{0}_2] \wedge [y\bar{+}x = \bar{0}_2]$	$(\forall_I Resclass\ L_{23})$
L_{25}.	$\vdash Inverse(\mathbb{Z}_2, \lambda x{\scriptscriptstyle\blacksquare} \lambda y{\scriptscriptstyle\blacksquare} x\bar{+}y, \bar{0}_2)$	$(\equiv_I\ L_4\ Inverse)$

Table 7.3: Proof with the `EquSolve` strategy.

with a proof of the example theorem from section 7.3.1 $Inverse(\mathbb{Z}_2, \lambda x{\scriptscriptstyle\blacksquare} \lambda y{\scriptscriptstyle\blacksquare} x\bar{+}y, \bar{0}_2)$, displayed in table 7.3.

The construction of the proof is in the beginning (lines L_{25} through L_{20}) nearly analogous to the one in the preceding section. The only exception is that no case splits are carried out after the applications of $\forall_I Resclass$ and $\exists_I Resclass$. Instead we get two equations in the lines L_{18} and L_{19}, which can be generally solved using the *SolveEqu* method. This method is applicable if MAPLE can compute a general solution to the given equation. In case the equation in question contains meta-variables the solution MAPLE computes is also used to constrain these meta-variables. In our example the meta-variable mv is substituted by c during the first application of *SolveEqu*. This is indicated by $\{mv \leftarrow c\}$ in the justification of line L_{18}. As already described in the last section this substitution is not carried out directly but added as a constraint on the meta-variable. Thus, the constraint on mv changes the formula implicitly in the remaining open goal L_{15} to $c \in \{0,1\}$, which can then be immediately closed with line L_2.

As opposed to the `TryAndError` strategy, the proofs `EquSolve` constructs are independent of the size of the residue class set involved. But the strategy can be applied successfully to only some of the possible occurring problems. In particular, neither problems involving the closure property nor refutations of a property can be tackled with `EquSolve`.

7.3.3 Applying Known Theorems

The motivation for our third strategy `ReduceToSpecial` is to incorporate the application of already proved theorems. The implementation of this strategy also serves as a case study for the application of Ω-ANTS to compute and suggest applicable assertions in parallel to the proof planning process as presented in chapter 4.1.2.

The very first call to the `ReduceToSpecial` strategy initializes Ω-ANTS with clusters for theorems from the theory of residue classes. Apart from the cluster for *Closed* theorems given in the example in chapter 4.1.2, we have clusters for theorems dealing with associativity, unit element, inverses, and divisors problems.

We observe the behavior of the `ReduceToSpecial` strategy with the proof for the theorem $Closed(\mathbb{Z}_5, \lambda x{\scriptscriptstyle\blacksquare} \lambda y{\scriptscriptstyle\blacksquare} (x\bar{*}y)\bar{+}\bar{3}_5)$ given in table 7.4. Among the theorems involved are those six theorems dealing with the closure property given in chapter 4.1.2. The overall approach of the `ReduceToSpecial` strategy is to always apply possible theorems with the *ApplyAss* method before considering any of its other methods. In case there is more than one applicable theorem at a time, `ReduceToSpecial` applies the first of these theorems and keeps the others for pos-

$L_1.$	$\vdash \bar{3}_5 \in \mathbb{Z}_5$	$(InResclSet)$
$L_2.$	$\vdash 5 \in \mathbb{Z}$	$(InInt)$
$L_3.$	$\vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare x)$	$(ApplyAss\ ClosdFV\ L_2)$
$L_4.$	$\vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare y)$	$(ApplyAss\ ClosedSV\ L_2)$
$L_5.$	$\vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare \bar{3}_5)$	$(ApplyAss\ ClosedConst\ L_2\ L_1)$
$L_6.$	$\vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare x \bar{*} y)$	$(ApplyAss\ ClComp\bar{*}\ L_2\ L_3\ L_4)$
$L_7.$	$\vdash Closed(\mathbb{Z}_5, \lambda x_\blacksquare \lambda y_\blacksquare (x \bar{*} y) \bar{+} \bar{3}_5)$	$(ApplyAss\ ClComp\bar{+}\ L_2\ L_5\ L_6)$

Table 7.4: Proof with the `ReduceToSpecial` strategy.

sible later backtracking.

The first step is the application of the *ClComp*$\bar{+}$ theorem as illustrated in chapter 4.1.2. This results in the three new subgoals L_6, L_5, and L_2. The former can be reduced applying the *ClComp*$\bar{*}$ theorem. For this the support agent of Ω-ANTS searching for premises of the assertion that already exists in the proof, finds line L_2 containing the sort information necessary to apply the theorem. Thus, the application of the *ClComp*$\bar{*}$ theorem introduces only two new subgoals, lines L_4 and L_3. These two lines can be closed by the application of two theorems involving a simple operation, namely *ClosedFV* and *ClosedSV*, again using line L_2 as the necessary premise. Both assertion applications are computed with the function agent responsible for matching theorems involving simple operations, only. Similarly, line L_5 is closed, applying theorem *ClosedConst*. However, this theorem has as additional premise that the constant congruence class occurring in the operation is in fact an element of the given residue class set. This premise is introduced as a new subgoal in line L_1.

After all possible applications of *ApplyAss* those remaining premises of the applied theorems that cannot be closed by theorem application have to be tackled. This can be done by some other methods associated with the strategy like *InInt* and *InResclSet*, which close goals of the form $n \in \mathbb{Z}$, if n is an integer, and $c \in RS_n$, if c is an element of the residue class set RS_n, respectively.

We have also experimented with bookkeeping of already solved problems and dynamically feeding them to Ω-ANTS and thereby extending the set of available theorems. However, this approach has the disadvantage that with many new theorems the parallel matching degenerates to a few parallel agents sequentially matching a lot of theorems. Furthermore, the application of theorems like the one from our example are not necessarily meaningful from a mathematical point of view and, more importantly, are rarely applicable in general and thus only cloak up our otherwise efficient mechanism to match theorems. Finally, it might also lead to proofs that are hard to replay in the same form. For instance, a theorem has just been proved in one run of ΩMEGA and is fed into Ω-ANTS as available theorems but not otherwise saved in the knowledge base. If it is then applied during the proof of another theorems within the same run of ΩMEGA, the proof of the latter theorem can then no longer be replayed in the same form in a different run of ΩMEGA if the former theorem is not available.

Like the `EquSolve` strategy `ReduceToSpecial` is independent of the size of the residue class set involved. Theoretically it is applicable to all types of problems. However, this depends on what kind of theorems are available in the knowledge base; that is, if for some problem no matching theorems are supplied these problems cannot be tackled. Likewise, if some intermediate theorems are missing some proofs with the `ReduceToSpecial` strategy cannot be concluded and the strategy will be backtracked.

7.3.4 Treating Direct Products

So far we have explained the strategies with residue class structures involving simple sets, only. In case the set involved consists of a direct product of single residue class sets, the proofs constructed by the EquSolve and the TryAndError strategy are slightly different. In fact, the only differences are the treatment of quantified variables that range over direct products and equations between tuples in proofs. They are transformed into a form that is suitable for the methods for simple residue class sets.

As an example we consider the set $\mathbb{Z}_2 \otimes \mathbb{Z}_2$ with the addition $\bar{+}$ and multiplication $\bar{*}$ as operations on the components. The proof works similar to the proofs given for the simple case of $\mathbb{Z}_2$ in sections 7.3.1 and 7.3.2. We omit to repeat all the details of these proofs and just describe the actual differences. The existential quantification

$$\exists z{:}\mathbb{Z}_2{\otimes}\mathbb{Z}_2{\scriptstyle\blacksquare}\,(cl_2(c_1), cl_2(c_2))\ [\bar{+}\times\bar{*}]\ z = (\bar{0}_2, \bar{0}_2)$$

is rewritten to

$$\exists x{:}\mathbb{Z}_2{\scriptstyle\blacksquare}\,\exists y{:}\mathbb{Z}_2{\scriptstyle\blacksquare}\,(cl_2(c_1), cl_2(c_2))\ [\bar{+}\times\bar{*}]\ (x, y) = (\bar{0}_2, \bar{0}_2),$$

for which the $\exists_I$*Resclass* is applicable. The resulting equation on tuples

$$(cl_2(c_1), cl_2(c_2))\ [\bar{+}\times\bar{*}]\ (cl_2(mv_1), cl_2(mv_2)) = (\bar{0}_2, \bar{0}_2)$$

is split into equations on the single components

$$cl_2(c_1)\bar{+}cl_2(mv_1) = \bar{0}_2 \quad \wedge \quad cl_2(c_2)\bar{*}cl_2(mv_2) = \bar{0}_2.$$

Universal quantification is treated analogously to existential quantification, inequalities on tuples result in the disjunction of inequalities on the elements of the tuples. These transformations are achieved by special methods, which are included in the strategies EquSolve and TryAndError.

7.4 Automatically Classifying Residue Class Sets

In order to obtain a large number of examples to test the strategies presented in the preceding section it seems appropriate to systematically construct theorems with respect to given residue class structures. The idea is to classify a given residue class structure in terms of the algebraic structure it forms by proving stepwise single properties; that is, we classify structures with one operation in terms of

1. magma, semi-group, quasi-group, monoid, loop, or group, and
2. whether a given structure is Abelian or not.

Structures with two operations are classified in terms of ring, ring-with-identity, division ring, or field. This section presents how the two classification processes work.

7.4.1 Classifying Structures with One Operation

The main idea of the classification of residue class structures is to check stepwise properties of the structure. This is done in three parts: First the likely answer to

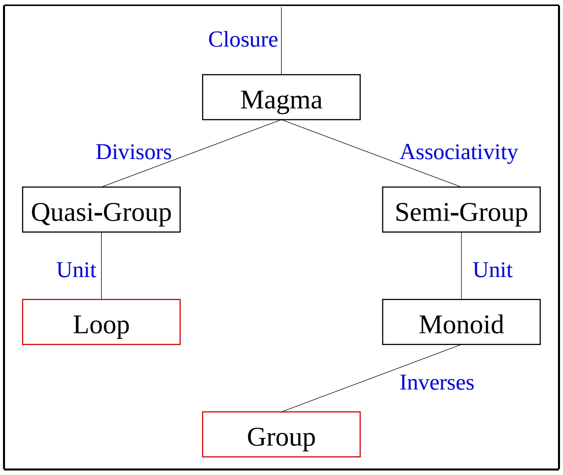


Figure 7.1: Classification schema for sets with one operation.

whether a certain property holds or not is computed using a computer algebra system. Depending on the result of this computation a proof obligation is constructed stating either that the property in question holds or that it does not hold. This proof obligation is then passed to the proof planner, which tries to discharge it immediately. If the proof fails the negated proof obligation is constructed and passed to the planner to prove the obligation. If both proving attempts fail the classification process stops and signals an error, otherwise the classification proceeds by checking the next property.

Properties are checked in a schematic order that eventually gives an answer to the question what kind of algebraic entity the input structure forms. We shall discuss here the classification schema for a given residue class set together with a single binary operation, which is displayed in figure 7.1. The classification process itself only produces proof obligations, which are subsequently discharged by constructing proof plans with Multi as described in section 7.3.

The first property we have to check is whether the given structure is actually closed under the operation. This is done with a multiplication table that is constructed in ΩMEGA. In case it can be proved that the structure is not closed the classification stops at this point. Otherwise we know that the structure in question forms a magma and the constructed multiplication table is passed to Gap. The classification proceeds then along the right branch of the schema in figure 7.1. The single tests given as labels on the edges (i.e., the test for associativity, whether there exists a unit element, and if all elements have an inverse with respect to the unit element) are performed using Gap. This way we show whether the given structure is a semi-group, a monoid or a group.

In case it turns out that the given structure is not associative the classification follows the left branch of the schema. Here the first test is to check whether the property of divisors holds. Since there is no appropriate algorithm in Gap we perform this test within ΩMEGA using the originally constructed multiplication table. If the divisors property can be successfully proved the structure forms at least a quasi-group. If the quasi-group contains additionally a unit element, which is again tested with Gap, it is a loop. If the structure forms a loop we do not have to check any further since we know that the structure is definitely not a group because we already proved it is non-associative.

To perform the test we can use the same functionality that we employ within the `select-instance` control rule during the proof planning process. For this the single functions are implemented to return two distinct values, namely a boolean value and a hint for the proof. For example, when checking whether in a given structure exist inverses for all elements, the appropriate function either returns true and a set of pairs of elements and inverses or it returns false and an element for which no inverse exists.

Once the classification with respect to the schema in figure 7.1 is finished and the structure in question is at least a magma, it is always checked whether it is Abelian. This test (i.e., whether or not the multiplication table is commutative) is again performed by GAP.

7.4.2 Classifying Structures with two Operations

So far we were only concerned with the classification of residue class sets together with one binary operation. But we can also automatically classify residue class sets together with two operations without much additional machinery.

A given structure of the form $(RS_n, \circ, \star)$ is first classified with respect to the first operation as described in section 7.4.1. If $(RS_n, \circ)$ is an Abelian group, we try to establish distributivity of $\star$ over $\circ$ corresponding to proving either of the theorems 7 in section 7.1. The proofs for distributivity are also planned by MULTI using exactly the same three strategies presented in section 7.3.

If distributivity can be successfully proved the residue class set is first reduced by the unit element of the first operation and the resulting set is then classified with respect to the second operation. More precisely, if e is the unit element in RS_n with respect to $\circ$, $(RS_n \backslash \{e\}, \star)$ is classified as described in the preceding sections. The result of this latter classification determines the exact nature of $(RS_n, \circ, \star)$, whether it is a ring, ring-with-identity, division ring, or field.

7.5 Experiments

The strategies presented in section 7.3 were designed on the basis of a relatively small number of examples. In detail, we used 21 examples to design the basic versions of the `ReduceToSpecial`, `TryAndError`, and `EquSolve` strategies (as described in section 7.3). For the extensions to handle direct products (section 7.3.4) we used 3 additional examples, and for the extensions to classify structures with two operations (section 7.4.2) we needed 2 examples, which were combinations of already used examples.

To guarantee that the presented approach is appropriate we tested the strategies against a large number of examples that differed from those used during the design process. In particular, we provide some evidence with our tests that

- our techniques realized in the strategies provide a machinery suitable to prove a large variety of problems about residue classes,
- the elaborate strategies we have developed are applicable to a sufficiently large number of examples, and
- the integration of computer algebra enhances indeed the proof planning process.

By gathering evidence for the latter two points we want to ensure that not a vast majority of problems is proved with an exhaustive case analysis using crude force search.

We created a testbed of roughly 13 million automatically generated examples and started to classify these examples. The examples are composed from the possible subsets of the residue classes modulo n, where n ranges between 2 and 10, together with operations that are systematically constructed from the basic operations. We tried to exclude repeating and trivial cases (e.g., sets with only one element or operations like $x - x$) as far as possible.

Although we tested the classification of structures with two operations on some examples, we focused on the classification of structures with one operation, because the results of the exploration of structures with one operation can ease the exploration of structures with two operations. Hence, we do not provide results about large experiments on structures with two operations.

We classified 14337 structures with one operation so far; the sets of these structures mainly involve $\mathbb{Z}_2$ to $\mathbb{Z}_6$ and some of their subsets. We found 5810 magmas, 109 Abelian magmas, 2064 semi-groups, 1670 Abelian semi-groups, 1018 quasi-groups, 461 Abelian quasi-groups, 93 Abelian monoids, and 780 Abelian groups (the other structures we tested are not closed). Among the classified structures are 45 structures with direct products. Note that these figures do not mean that we have so many distinct algebraic entities, since our sets contain many isomorphic structures. For the proofs of the single properties that were tested during the classification, MULTI employed successfully `ReduceToSpecial` to a sample of 19% and `EquSolve` to a different set accounting for 21% of the examples. The remaining 60% of the examples could be solved only by the `TryAndError` strategy. All input structures were successfully classified and the classification algorithm did not signal a single error.

The experiments demonstrate that our strategies are general enough to cope with a large number of problems. Although the percentage of problems during the classification of structures with one operation that could only be solved with `TryAndError` seems high at a first glance, it is actually not so disappointing. We have to take into account that nearly all proofs involving the closure property of non-complete residue class sets (i.e., sets such as $\mathbb{Z}_3\backslash\{\bar{1}_3\}$) and the refutation of properties could only be solved with the `TryAndError` strategy. Thus, our tests show that the more elaborate strategies are indeed applicable on a considerable number of examples.

7.6 Summary of Chapter 7

The case study presented in this chapter was an experiment in exploring simple properties of residue classes over integers. Single properties are step-wise checked by an exploration module that generates the appropriate proof obligations. These proof obligations are passed to the multi-strategy proof planner MULTI to be proved. The proof planner can draw on three different planning strategies in order to solve the problems, where each strategy is an implementation of one aspect of the combination of proof planning with both computer algebra and Ω-ANTS. We employ GAP to guide the search process via control rules, MAPLE is applied directly with methods in order to justify equational goals, and assertion applications are computed with Ω-ANTS when reusing existing theorems.

We gave some empirical evidence that the currently implemented methods form a robust set of planning operators that suffices to explore the domain of residue classes. Moreover, the case study shows how a combination of various systems can be fruitfully employed to large classes of examples. Although one major ingredient in our setup is the multi-strategy proof planner MULTI, the presented work does not reflect the full power of proof planning in general and its multi-strategy variant in particular, for two reasons: Firstly, the problem solutions are still quite algorithmic and the necessary search is rather limited. And secondly, problems can be solved with a single strategy each, and therefore the case study does not show the additional power one gains from interleaving strategies, which is the subject of the following chapter.

Chapter 8

Isomorphism Proofs

The last chapter provided a case study proving simple properties of residue classes and a classification of residue class structures in terms of the algebraic entity they form. The case study was designed for the demonstration of the combination of Multi and Ω-Ants and the use of computer algebra in proof planning both in method application and control rules. However, from a proof planning perspective the proofs were relatively straightforward; in particular, single strategies were generally able to prove a whole problem.

In this chapter we shall now examine a class of problems whose solution require the full power of multi-strategy proof planning. In detail, we shall reuse the same, albeit slightly extended, strategies developed for the preceding case study in order to prove that two given residue class structures are either isomorphic or not isomorphic to each other. For the construction of the proofs we rely on the combination and interleaving of different strategies. Thus, the case study presents full-scale multi-strategy proof planning together with the different combinations of proof planning with computer algebra and Ω-Ants.

Moreover, we shall further exploit our results from the preceding chapter and extend our automatic exploration of the residue class domain. The exploration presented in chapter 7.4 returns sets of magmas, abelian magmas, semi-groups, etc. This, however, does not indicate how many of these structures are actually different (i.e., not isomorphic to each other) or are just different representations of the same structure. The proof techniques we present in this chapter enable us to further classify residue class structures by dividing them into isomorphism classes. The classification into isomorphism classes has been reported in [151]; a complete report on the exploration of the residue class domain can be found in [150].

This chapter is structured as follows: We first introduce the problems and their formalizations. We then describe how both isomorphism and non-isomorphism proofs are planned and point out the peculiarities when residue class structures with direct products are involved. Finally, we present the algorithm to automatically classify residue class structures into isomorphism classes before detailing the experiments we have carried out.

8.1 Problems and Formalization

Since we are interested in distinguishing classes of isomorphic residue class structures the problems we consider state that two residue class structures $(RS^1_{n_1}, \circ_1)$ and $(RS^2_{n_2}, \circ_2)$ are isomorphic or not isomorphic to each other. Thus, the two relevant

(a) $Iso(RS^1_{n_1}, \circ_1, RS^2_{n_2}, \circ_2)$	(b) $\neg Iso(RS^1_{n_1}, \circ_1, RS^2_{n_2}, \circ_2)$

Table 8.1: Isomorphism theorems for residue class structures.

theorems are those given in table 8.1.

To state the isomorphism theorems we have to define the Iso predicate, which is formalized in a straightforward way:

$$\begin{aligned} Iso \;\equiv\; & \lambda A_{\alpha o} \centerdot \lambda \circ_{\alpha\alpha\alpha} \centerdot \lambda B_{\beta o} \centerdot \lambda \star_{\beta\beta\beta} \centerdot \\ & \exists h{:}_{F(A,B)} \centerdot Inj(h, A) \wedge Surj(h, A, B) \wedge Hom(h, A, \circ, B, \star) \end{aligned} \tag{8.1}$$

Definition (8.1) simply states the fact that two structures $(A, \circ)$ and $(B, \star)$ are isomorphic if and only if there exists a function $h : A \to B$, such that h is an injective and surjective homomorphism. The sort $F(A, B)$ of h is the set of all total functions from A into B. The concepts of injective, surjective and homomorphism were already defined in chapter 6.2.

8.2 Isomorphism Proofs

In this section we present how Multi plans isomorphism proofs. It employs the same strategies already described in chapter 7.3, which require only few supplemental methods in addition to those needed to prove single properties of residue class sets. We just had to add two methods for the introduction of isomorphism mappings to the `TryAndError` and `EquSolve` strategies and one additional theorem for the `ReduceToSpecial` strategy. Contrary to the proofs in chapter 7.3 that could be solved in most cases within one strategy, for isomorphism proofs different subproofs can be solved by different strategies. In detail this means, that the strategy `EquSolve` switches to `TryAndError`, while `ReduceToSpecial` uses `EquSolve` and `TryAndError` to prove some of the occurring subgoals.

8.2.1 `TryAndError`

For the proof that two given structures are isomorphic we have to find a mapping that is a bijective homomorphism. In the context of finite sets each possible mapping can be represented as a pointwise defined function, where the image of each element of the domain is explicitely specified as an element of the codomain. Following the idea of the strategy `TryAndError`, a case analysis for the different possibilities for defining the mapping is performed. If it cannot be shown that the mapping is a homomorphism or a bijection, the next mapping is constructed and verified.

Table 8.2 shows the first steps of the proof for the claim that $(\mathbb{Z}_2, \bar{+})$ is isomorphic to $(\mathbb{Z}_3 \backslash \{\bar{0}\}, \bar{*})$. To preserve space we have omitted some of the less interesting details and abbreviated some of the hypotheses lists.

The topmost case split (i.e., the case split over the possible instantiations of the isomorphism mapping) is introduced with the application of the $\exists_I ResclFunc$ method in line L_{98}. $\exists_I ResclFunc$ introduces a constant h' for the existentially quantified variable h, which represents a function from $\mathbb{Z}_2$ to $\mathbb{Z}_3 \backslash \{0\}$. This function is also explicitly introduced in line L_1 as the formalization of a pointwise function

$$h' : \mathbb{Z}_2 \longrightarrow \mathbb{Z}_3 \setminus \{\bar{0}_3\} \quad \text{with} \quad h'(x) = \begin{cases} cl_3(mv_1), & \text{if } x = \bar{0}_2 \\ cl_3(mv_2), & \text{if } x = \bar{1}_2 \end{cases},$$

$L_1.$	L_1	$\vdash h' = \lambda x{\scriptstyle\blacksquare}{}^{\imath}y{\scriptstyle\blacksquare}[x = \bar{0}_2 \Rightarrow y = cl_3(mv_1)] \wedge$ $[x = \bar{1}_2 \Rightarrow y = cl_3(mv_2)]$	(Hyp)
		$\vdots$	
$L_5.$	L_5	$\vdash cl_2(c_1) \in \mathbb{Z}_2$	(Hyp)
$L_6.$	L_6	$\vdash cl_2(c_2) \in \mathbb{Z}_2$	(Hyp)
		$\vdots$	
$L_{10}.$	L_{10}	$\vdash c_1 = 0$	(Hyp)
$L_{11}.$	L_{11}	$\vdash c_2 = 1$	(Hyp)
		$\vdots$	
$L_{70}.$	$\mathcal{H}_3$	$\vdash 1 \neq 2$	$(\neq Reflexivity)$
$L_{71}.$	$\mathcal{H}_3$	$\vdash 1 \neq 2 \vee 0 = 1$	$(\vee_{Il}\ L_{70})$
$L_{72}.$	$\mathcal{H}_3$	$\vdash cl_3(mv_1) \neq cl_3(mv_2) \vee 0 = 1_{\{mv_1 \leftarrow 1, mv_2 \leftarrow 2\}}$	$(ConCongCl\ L_{71})$
$L_{73}.$	$\mathcal{H}_3$	$\vdash h'(\bar{0}_2) \neq h'(\bar{1}_2) \vee 0 = 1$	$(AppFct\ L_1\ L_{72})$
$L_{74}.$	$\mathcal{H}_3$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee c_1 = c_2$	$(SimplNum\ L_{10}\ L_{11}\ L_{73})$
$L_{75}.$	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee c_1 = c_2$	$(\vee E{*}{*}\ L_5\ L_6 L_{74} \ldots)$
$L_{76}.$	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee cl_2(c_1) = cl_2(c_2)$	$(ConCongCl\ L_{75})$
$L_{77}.$	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) = h'(cl_2(c_2)) \Rightarrow cl_2(c_1) = cl_2(c_2)$	$(\vee 2\Rightarrow\ L_{76})$
$L_{78}.$	$\mathcal{H}_1$	$\vdash \forall y{:}\mathbb{Z}_2{\scriptstyle\blacksquare} h'(cl_2(c_1)) = h'(y) \Rightarrow cl_2(c_1) = y$	$(\forall_I Resclass\ L_{77})$
$L_{79}.$	L_1	$\vdash \forall x{:}\mathbb{Z}_2, y{:}\mathbb{Z}_2{\scriptstyle\blacksquare} h'(x) = h'(y) \Rightarrow x = y$	$(\forall_I Resclass\ L_{78})$
$L_{80}.$	L_1	$\vdash Inj(h', \mathbb{Z}_2)$	$(\equiv_I\ L_{79}\ Inj)$
		$\vdots$	
$L_{96}.$	L_1	$\vdash mv_1 \in \{1,2\} \wedge mv_2 \in \{1,2\}$	$(\wedge_I \ldots)$
$L_{97}.$	L_1	$\vdash Inj(h', \mathbb{Z}_2) \wedge Surj(h', \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\})$ $\wedge Hom(h', \mathbb{Z}_2, \lambda xy{\scriptstyle\blacksquare} x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy{\scriptstyle\blacksquare} x \bar{*} y)$	$(\wedge_I \ldots)$
$L_{98}.$		$\vdash \exists h{:}_{F(\mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\})}{\scriptstyle\blacksquare} Inj(h, \mathbb{Z}_2) \wedge Surj(h, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\})$ $\wedge Hom(h, \mathbb{Z}_2, \lambda xy{\scriptstyle\blacksquare} x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy{\scriptstyle\blacksquare} x \bar{*} y)$	$(\exists_I ResclFunc\ L_{96}\ L_{97})$
$L_{99}.$		$\vdash Iso(\mathbb{Z}_2, \lambda xy{\scriptstyle\blacksquare} x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy{\scriptstyle\blacksquare} x \bar{*} y)$	$(\equiv_I\ L_{98}\ Iso)$

$$\mathcal{H}_1 = \{L_1, L_5\},\ \mathcal{H}_2 = \{L_1, L_5, L_6\},\ \mathcal{H}_3 = \{L_1, L_5, L_6, L_{10}, L_{11}\}$$

Table 8.2: Introduction of the pointwise defined function.

where the mv_i are meta-variables that can be instantiated by elements of the range, in our example by 1 or 2. These possible instantiations are also introduced by $\exists_I ResclFunc$ in line L_{96}. We can now search for an appropriate combination of mv_1 and mv_2, or in other words over all possible functions h' for which the properties given in line L_{97} hold.

Multi abbreviates the search for the right function h' by computing a hint. For an isomorphism $h{:}(RS^1_n, \circ_1) \to (RS^2_m, \circ_2)$, Maple is asked to give a solution for the system of equations $x_k = x_i \circ_2 x_j$ with respect to the modulo factor m using Maple's function `msolve`. The system is generated by instantiating the homomorphism equation $h(cl_n(k)) = h(cl_n(i)) \circ_2 h(cl_n(j))$, where $cl_n(k) = cl_n(i) \circ_1 cl_n(j)$ for all values $cl_n(i), cl_n(j) \in RS^1_n$. Thus, $h(cl_n(l))$ becomes the variable x_l in our equation system. When Maple returns a solution for the variables containing only elements from the integer set corresponding to RS^2_m we have found a homomorphism between the structures. When there is a disjoint solution with $x_i \neq x_j$, for all $i \neq j$, we have a candidate for the isomorphism.

In the example in figure 8.2 Maple is asked to give a solution for the equations $x_0 = x_0 * x_0$, $x_1 = x_0 * x_1$, $x_1 = x_1 * x_0$, $x_0 = x_1 * x_1$ with modulo factor 3 and returns $\{x_1 = 0, x_0 = 0\}$, $\{x_1 = 2, x_0 = 1\}$, $\{x_0 = 1, x_1 = 1\}$. The solutions are analyzed by the hint system, and the second is suggested because it is both a disjoint solution and all elements are in the codomain. Therefore, $h'(\bar{0}_2) = \bar{1}_3$, $h'(\bar{1}_2) = \bar{2}_3$ is inserted as the pointwise defined isomorphic function (by adding the instantiations $\{mv_1 \leftarrow 1, mv_2 \leftarrow 2\}$ as displayed in line L_{72}).

We observe how the function h' is applied in the subproof for injectivity in figure 8.2 beginning with line L_{80} backwards. The proof until L_{73} is the fairly standard procedure of the `TryAndError` strategy: Defined concepts are expanded, quantifiers are eliminated by introducing case splits and statements about residue classes are rewritten into statements about integers. The interesting part is then the application of the *AppFct* method in line L_{73}. This corresponds to the substitution of the functional expressions given on the righthand side of the disjunction in line L_{73} with the functional values given in the definition of h' in line L_1. The result is given in line L_{72}. The rest of the subproof can then be easily concluded.

For a given function h' Multi has to construct subproofs of n^2 cases for each of the three properties that h' is surjective, injective, and a homomorphism. Here n is the cardinality of the structures involved. However, if no suitable hint can be computed there are n^n pointwise defined functions to check, which becomes infeasible already for relatively small n.

8.2.2 EquSolve

During the isomorphism proof we have to show that the introduced mapping is a bijective homomorphism. Doing so by a complete case analysis can become quite lengthy and therefore it is desirable to represent the isomorphism function in a more compact form. Often this can be realized by computing a polynomial that interpolates the pointwise defined function. If we can compute such an interpolation polynomial the `EquSolve` strategy has a chance of finding the subproofs for surjectivity and the homomorphism property. Note that in the subproof for injectivity we still have to show for any two distinct elements that their images differ, which cannot be done with the `EquSolve` strategy.

For the construction of the interpolation polynomial we again employ Maple. However, we do not use any of the standard algorithms for interpolating sparse polynomials from the literature (see for example [223, 224, 221]) as they do not necessarily give us an interpolation polynomial, which is optimal for our purposes. Moreover, some of the implemented interpolation algorithms, for instance in Maple, do not always suffice for our purposes.[1] This is especially true for the case of multivariate polynomial interpolation that is necessary for dealing with residue class sets that are composed of direct products, which we shall describe in more detail in section 8.4. We have thus decided to implement a simple search algorithm to find a suitable interpolation polynomial of minimal degree. This is feasible as we have to handle only relatively small mappings.

In detail, the interpolation proceeds as follows: Given a pointwise defined isomorphism function $h{:}cl_n(x_i){\in}RS^1_n \to cl_m(y_i){\in}RS^2_m$ the algorithm iteratively constructs systems of equations $(a_d x_i^d + \cdots + a_1 x_i + a_0)\ mod\ m = y_i\ mod\ m$ for all x_i, y_i for $d = 0, 1, \ldots$. These equations are sent to Maple to solve them with the regular `msolve` function. In case Maple returns a solution for $a_d, \ldots, a_0$ we have found an interpolating polynomial. Otherwise a system of polynomials with degree $d+1$ is sent to Maple. This procedure terminates at the latest when $d = m-1$. Thus, instead of using Maple's interpolation algorithm directly we only use Maple to solve the given systems of equations and assemble the interpolation polynomial separately.

We illustrate this for the proof that $(\mathbb{Z}_2, \lambda xy_{\bullet} x \bar{+} y \bar{+} \bar{1}_2)$ is isomorphic to $(\mathbb{Z}_2, \bar{+})$ shown in figure 8.3. The corresponding pointwise isomorphism mapping is $h(\bar{0}_2) = \bar{1}_2, h(\bar{1}_2) = \bar{0}_2$ for which the interpolation polynomial $x \to x + 1\ mod\ 2$ can be

[1] Maple's algorithms `interp` and `Interp` cannot always handle the interpolation of functions where a non-prime modulo factor is involved.

L_5.	L_5	$\vdash cl_2(c) \in \mathbb{Z}_2$	(Hyp)
		$\vdots$	
L_{91}.	L_5	$\vdash (mv_y + 1)\ mod\ 2 = c\ mod\ 2_{\{mv_y \leftarrow c-1\}}$	$(SolveEqu)$
L_{92}.	L_5	$\vdash cl_2(mv_y)\bar{+}\bar{1}_2 = cl_2(c)$	$(ConResclSet\ L_{91})$
L_{93}.	L_5	$\vdash mv_y \in \{0,1\}$	$(Open)$
L_{94}.	L_5	$\vdash \exists y{:}\mathbb{Z}_2\text{.}\, y\bar{+}\bar{1}_2 = c$	$(\exists_I\, Resclass\ L_{92}\ L_{93})$
L_{95}.		$\vdash \forall x{:}\mathbb{Z}_2\text{.}\, \exists y{:}\mathbb{Z}_2\text{.}\, y\bar{+}\bar{1}_2 = x$	$(\forall_I\, Resclass\ L_{94})$
L_{96}.		$\vdash Surj(\lambda x\text{.}\, x\bar{+}\bar{1}_2, \mathbb{Z}_2, \mathbb{Z}_2)$	$(\equiv_I\ L_{95}\ Surj)$
L_{97}.		$\vdash Inj(\lambda x\text{.}\, x\bar{+}\bar{1}_2, \mathbb{Z}_2)$	$(Open)$
L_{98}.		$\vdash Hom(\lambda x\text{.}\, x\bar{+}\bar{1}_2, \mathbb{Z}_2, \lambda xy\text{.}\, x\bar{+}y, \mathbb{Z}_2, \lambda xy\text{.}\, x\bar{+}y\bar{+}\bar{1}_2)$	$(Open)$
L_{99}.		$\vdash Iso(\mathbb{Z}_2, \lambda xy\text{.}\, x\bar{+}y, \mathbb{Z}_2, \lambda xy\text{.}\, x\bar{+}y\bar{+}\bar{1}_2)$	$(IsoPoly_I\ L_{96}\ L_{97}\ L_{98})$

Table 8.3: Introduction of the interpolated function.

computed. The strategy EquSolve applies the method *IsoPolyI* on the isomorphism statement in line L_{99}. One of the application conditions of *IsoPolyI* is that a suitable interpolation polynomial can be computed. If successful, the application of the method introduces the polynomial as isomorphism mapping into the proof. The single properties *Surj*, *Inj*, and hom given in lines L_{96}, L_{97}, and L_{98} respectively, have then to be shown for the interpolation polynomial. In figure 8.3 we have only carried out the details for the subproof of surjectivity, in which the problem is reduced to an equality over integers that can be generally solved by MAPLE employing the *Solve−Equ* method similar to the proof in section 7.3.2. The proof of the homomorphism property proceeds analogously. The proof for injectivity, however, cannot be constructed with the EquSolve strategy for the reasons already explained. Therefore, to close the subgoal in line L_{97} MULTI switches either to the strategy ReduceToSpecial or TryAndError. How the former is applied in this context is described in the next section. In case the latter strategy is applied the case analysis is conducted with the interpolation polynomial instead of the pointwise function as in section 8.2.1.

The success of EquSolve depends on the capabilities of MAPLE. Often equations in isomorphism proofs contain terms with different modulo factors nested inside, resulting from the mapping between residue class sets RS_n and RS_m with $n \neq m$, which are not solvable by MAPLE. So EquSolve is limited to proofs for residue class sets with the same modulo factor.

8.2.3 Using ReduceToSpecial

The strategic control rules in MULTI specify that on residue class problems the strategies ReduceToSpecial, EquSolve, and TryAndError are always tested in this order. This holds for isomorphism or non-isomorphism problems as well as for possible arising subproblems such as to show injectivity, surjectivity, or homomorphy. For instance, if EquSolve can introduce a suitable polynomial function but fails to prove the arising injectivity, surjectivity, or homomorphy subgoals, MULTI has to deal with those subproblems again on the strategic level. Since we do not have theorems to handle isomorphism problems in general (we only have one, in case the structures contain direct products, which is explained in section 8.4), ReduceToSpecial is not applicable to the original theorem, but it comes into play when a subgoal, in particular the injectivity subgoal, has to be proved. Here we can exploit the simple mathematical fact that in finite domains surjectivity implies injectivity and vice versa with the following theorem:

> *A surjective mapping between two finite sets with the same cardinality is injective.*

Thus, the proof of injectivity can be completely avoided, if we can prove that our mapping is surjective and that the structures are of the same cardinality. We have chosen this theorem rather than its dual, where we can infer that a mapping on finite structures is surjective if we already know that it is injective, since for the injectivity proof Multi always has to perform a case analysis. Hence, the idea for the most efficient isomorphism proofs is to start with `EquSolve` on the whole isomorphism problem, prove the surjectivity and homomorphy subproblem if possible with equational reasoning and, since `EquSolve` always fails on the injectivity subgoal, to let `ReduceToSpecial` finish the proof. Note that Multi's interleaving of strategies also allows for `ReduceToSpecial` to close the surjectivity subgoal while injectivity is not yet proved.

8.3 Non-Isomorphism Proofs

In this section we present how Multi can prove that two given structures are not isomorphic to each other, which corresponds to problem (b) from table 8.1. These proofs are essential since in case the isomorphism proof fails it is not necessarily the case that two structures are not isomorphic. If the two structures involved are of different cardinality they are trivially not isomorphic. This case is easily planned with the `ReduceToSpecial` strategy and an appropriate theorem. We shall not give the implementation of this case in detail and instead concentrate on the more interesting case where the two structures are of the same cardinality. For the proofs of this latter case we have implemented the following three proof techniques:

1. We show that each possible mapping between the two structures involved is not isomorphic. This is again an exhaustive case analysis for which we employ the slightly extended `TryAndError` strategy.
2. We argue that one of the structures contains an argument of an order or a substructure of a certain cardinality that is not reflected in the other structure. (We shall define the necessary notions later in this chapter.) This technique is achieved by interleaving the `ReduceToSpecial` and `TryAndError` strategies.
3. We construct a contradiction by assuming first there exists an isomorphism between the two residue class structures and deriving then that it is not injective. For this technique we have implemented a new strategy, called `NotInjNotIso`.

We shall explain this new strategy as well as the extensions to the already introduced strategies using the example that the two abelian semi-groups $(\mathbb{Z}_4, \lambda xy_{\bullet} x \bar{*} y \bar{*} \bar{2}_4)$ and $(\mathbb{Z}_4, \lambda xy_{\bullet} \bar{2}_4)$ are not isomorphic.

8.3.1 Employing `TryAndError` Directly

As already stated in section 7.3.1 the two basic principles of the `TryAndError` strategy are to resolve quantified statements over finite domains by checking all possible cases or alternatives and to rewrite statements on residue classes into corresponding statements on integers. When solving non-isomorphism problems the top-most case split is to check for each possible function from the one residue class set into the other that it is either not injective, not surjective, or not a homomorphism.

$L_1.$	L_1	$\vdash h' = \lambda x_\bullet \iota y_\bullet [x = \bar{0}_4 \Rightarrow y = cl_4(c_1)] \wedge [x = \bar{1}_4 \Rightarrow y = cl_4(c_2)] \wedge$ $[x = \bar{2}_4 \Rightarrow y = cl_4(c_3)] \wedge [x = \bar{3}_4 \Rightarrow y = cl_4(c_4)]$	(Hyp)
$L_2.$	L_2	$\vdash c_1 \in \{0,1,2,4\}$	(Hyp)
$L_3.$	L_3	$\vdash c_2 \in \{0,1,2,4\}$	(Hyp)
$L_4.$	L_4	$\vdash c_3 \in \{0,1,2,4\}$	(Hyp)
$L_5.$	L_5	$\vdash c_4 \in \{0,1,2,4\}$	(Hyp)
$L_6.$	L_6	$\vdash c_1 = 0$	(Hyp)
$L_7.$	L_7	$\vdash c_2 = 0$	(Hyp)
$L_8.$	L_8	$\vdash c_3 = 0$	(Hyp)
$L_9.$	L_9	$\vdash c_4 = 0$	(Hyp)
$L_{10}.$	L_{10}	$\vdash c_1 = 1$	(Hyp)
		$\vdots$	
$L_{75}.$	$\mathcal{H}_3$	$\vdash \neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4))$	$(\vee_{Ir}\ L_{74})$
		$\vdots$	
$L_{95}.$	$\mathcal{H}_2$	$\vdash \neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(\vee_{Il}\ L_{94})$
$L_{96}.$	$\mathcal{H}_1$	$\vdash \neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(\vee_E^{**}\ L_2\ L_3\ L_4\ L_5$ $L_{95}\ L_{75}\ \ldots)$
$L_{97}.$		$\vdash \forall h{:}F(\mathbb{Z}_4,\mathbb{Z}_4)_\bullet \neg Inj(h, \mathbb{Z}_4) \vee \neg Surj(h, \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h, \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(\forall_I ResclFunc\ L_{96})$
$L_{98}.$		$\vdash \neg \exists h{:}F(\mathbb{Z}_4,\mathbb{Z}_4)_\bullet Inj(h, \mathbb{Z}_4) \wedge Surj(h, \mathbb{Z}_4, \mathbb{Z}_4) \wedge$ $Hom(h, \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(PullNeg\ L_{97})$
$L_{99}.$		$\vdash \neg Iso(\mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(\equiv_I\ L_{98}\ Iso)$

$\mathcal{H}_1 = \{L_1, L_2, L_3, L_4, L_5\},\ \mathcal{H}_2 = \mathcal{H}_1 \cup \{L_6, L_7, L_8, L_9\},\ \mathcal{H}_3 = \mathcal{H}_1 \cup \{L_7, L_8, L_9, L_{10}\}$

Table 8.4: Proof with the `TryAndError` strategy.

Table 8.4 shows the abbreviated proof for our non-isomorphism problem as it is constructed when applying `TryAndError`. In particular, we have renumbered the lines in order to preserve space. The proof works in the following way: After expanding the definition of isomorphism in line L_{99} the application of the method *PullNeg* pushes the negation to the inner-most formulas. Next $\forall IResclFunc$ is applied, a method for the elimination of universally quantified goals that is the dual of the $\exists IResclFunc$ method introduced in section 8.2. $\forall IResclFunc$ instantiates the variable h, a mapping between the two given residue class sets, with a constant h' and also introduces the hypotheses L_1 through L_5. L_1 explicitly states the function h' as a unary function mapping the elements of the domain to constants $cl_4(c_1)$ to $cl_4(c_4)$ of the codomain. The lines L_2 through L_5 contain the possible instantiations for the constants c_1, c_2, c_3, and c_4. The next step is then the case split over all possible mappings between the residue class sets, respectively all possible combinations of constants c_1 to c_4. It is introduced by the application of $\vee E{*}{*}$ with respect to the lines L_2 through L_5 to line L_{96}. The case split leads actually to 256 new open subgoals (i.e., all possible instantiations of h') where we have only given two (i.e., lines L_{95} and L_{75} in figure 8.4). Likewise we have given only a subset of the newly introduced hypotheses containing the different combinations of the constants c_1 to c_4. Each of the new subgoals considers a different combination of these constants in its hypotheses. It now remains to show for each case that the function represented by L_1 and the actual hypotheses is either not surjective, not injective, or not a homomorphism. For line L_{95}, for example, it can be easily shown that the mapping is not injective since all the images are $\bar{0}_4$.

The application of this naive technique suffers from combinatorial explosion on the possibilities for the function h. For two structures whose sets have cardinality n we have to consider n^n different possible functions. Thus, in practice this strategy is not feasible if structures of cardinality larger than four are involved. Despite this fact the strategy is our fall back if the other techniques presented in the sequel should fail.

8.3.2 Combining `ReduceToSpecial` and `TryAndError`

If two structures are isomorphic, they share the same algebraic properties. Thus, in order to show that two structures are not isomorphic it suffices to show that one particular property holds for one structure but not for the other. In this subsection we discuss two such properties and explain how Multi combines the strategies `ReduceToSpecial` and `TryAndError` to establish that two structures are not isomorphic. Thereby `ReduceToSpecial` employs theorems that can reduce the original goal to subgoals stating that a property does not hold for the one structure whereas it holds for the other structure. These subgoals can then be proved with `TryAndError`.

First we introduce the concepts *order, trace*, and *order of the trace* of elements of a structure $(S, \circ)$ where S is finite:

- An element $a \in S$ has the *order* n if $n \in \mathbb{N}$ is the smallest positive integer such that $a^n = \underbrace{a \circ \ldots \circ a}_{\text{n-times}} = e$, where $e \in S$ is the unit element with respect to $\circ$. In the following we write this as $Order(a)$.
- The *trace* of an element $a \in S$ is the set $\{a^n | n \in \mathbb{N}\}$. The cardinality of this set is referred to as the *order of the trace* of a. This is written as $OrderTr(a)$ in the following.

The latter concept is a generalization of the former so we can also deal with elements that do not have an order or with structures who do not have a unit element. Note also that both the order of an element a and the order of its trace always range between 1 and the cardinality of S.

For two structures $(S_1, \circ_1)$ and $(S_2, \circ_2)$ we can show that if they are isomorphic then for each element $a_1 \in S_1$ with order n there exists an element $a_2 \in S_2$ with the same order. An analogous statement can be proved for the order of the traces. Thus, to prove that two structures are not isomorphic it is sufficient to prove that one structure contains an element a_1 such that the other structure contains no element a_2 whose order (order of the trace) is equal to the order (order of the trace) of a_1. This can be formalized in the following theorems:

$$Ord\textbf{:}\ (\exists n{:}_{[1,Card(S_1)]}\blacksquare(\exists x_1{:}_{S_1}\blacksquare Order(x_1, S_1, \circ_1) = n) \wedge \\ (\neg\exists x_2{:}_{S_2}\blacksquare Order(x_2, S_2, \circ_2) = n)) \Rightarrow \neg Iso(S_1, \circ_1, S_2, \circ)$$

$$OrdTr\textbf{:}\ (\exists n{:}_{[1,Card(S_1)]}\blacksquare(\exists x_1{:}_{S_1}\blacksquare OrderTr(x_1, S_1, \circ_1) = n) \wedge \\ (\neg\exists x_2{:}_{S_2}\blacksquare OrderTr(x_2, S_2, \circ_2) = n)) \Rightarrow \neg Iso(S_1, \circ_1, S_2, \circ)$$

Here $[1, Card(S_1)]$ is the integer interval from 1 to the cardinality of S_1.

The `ReduceToSpecial` strategy can apply these two theorems to reduce non-isomorphic goals and then `TryAndError` takes over to complete the proof. Figure 8.5 displays the proof for our example. Here the application of the $OrdTr$ theorem on the goal in line L_{66} results in the new line L_{65}. The sort of the existentially quantified variable n in line L_{65} is an integer interval ranging from 1 to the cardinality of $\mathbb{Z}_4$. This variable is eliminated with an application of the $\exists_I Sort$ method. This method is a domain independent method that can generally deal with sorted existentially quantified goals. Similar to $\exists_I Resclass$ it splits the goal into two new

$L_1.$	L_1	$\vdash cl_4(c) \in \mathbb{Z}_4$	(Hyp)
$L_2.$	L_1	$\vdash c \in \{0,1,2,3\}$	$(ConResclSet\ L_1)$
		$\vdots$	
$L_{43}.$	L_1	$\vdash \neg OrderTr(cl_4(c), \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4) = 3$	$(\vee_E^{**} \ldots)$
$L_{44}.$		$\vdash \forall x_2{:}\mathbb{Z}_4 \centerdot \neg OrderTr(x_2, \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4) = 3$	$(\forall_I Resclass\ L_{43})$
$L_{45}.$		$\vdash \neg \exists x_2{:}\mathbb{Z}_4 \centerdot OrderTr(x_2, \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4) = 3$	$(PullNeg\ L_{44})$
		$\vdots$	
$L_{46}.$		$\vdash 1 = 1$	$(=Reflexivity)$
$L_{47}.$		$\vdash 1 \in \{1,2,3\}$	$(\vee_{Ir}\ L_{46})$
$L_{48}.$		$\vdash mv_2 \in \{0,1,2,3\}$	$(\vee_{Ir}\ L_{47})$
		$\vdots$	
$L_{53}.$		$\vdash Card(\{\bar{1}_4, \bar{2}_4, \bar{0}_4\}) = 3$	$(Open)$
$L_{54}.$		$\vdash OrderTr(cl_4(mv_2), \mathbb{Z}_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4){=}3_{\{mv_2 \leftarrow 1\}}$	$(RewrTrace\ L_{53})$
$L_{55}.$		$\vdash \exists x_1{:}\mathbb{Z}_4 \centerdot OrderTr(x_1, \mathbb{Z}_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4){=}3$	$(\exists_I Resclass\ L_{54}\ L_{48})$
$L_{56}.$		$\vdash [\exists x_1{:}\mathbb{Z}_4 \centerdot OrderTr(x_1, \mathbb{Z}_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4){=}mv_1]$ $\wedge[\neg \exists x_2{:}\mathbb{Z}_4 \centerdot OrderTr(x_2, \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4){=}mv_1]_{\{mv_1 \leftarrow 3\}}$	$(\wedge_I\ L_{55}\ L_{45})$
		$\vdots$	
$L_{64}.$		$\vdash mv_1 \in [1, Card(\mathbb{Z}_4)]$	$(Open)$
$L_{65}.$		$\vdash \exists n{:}[1, Card(\mathbb{Z}_4)] \centerdot [\exists x_1{:}\mathbb{Z}_4 \centerdot OrderTr(x_1, \mathbb{Z}_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4) = n]$ $\wedge\ [\neg \exists x_2{:}\mathbb{Z}_4 \centerdot OrderTr(x_2, \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4) = n]$	$(\exists_I Sort\ L_{56}\ L_{64})$
$L_{66}.$		$\vdash \neg Iso(\mathbb{Z}_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4)$	$(ApplyAss\ OrdTr\ L_{65})$

Table 8.5: Proof with the `TryAndError` and `ReduceToSpecial` strategies.

subgoals: The original goal formula where the variable is now instantiated and a subgoal containing the sort information. These correspond to lines L_{56} and L_{64} in our example. $\exists_I Sort$ also introduces a meta-variable, mv_1, for the witness term, but does not carry out any domain specific reformulations of the new goals or the sort information. In fact, its application is postponed by control rules if there is a more domain specific method applicable to deal with the quantifier. For instance is the application of $\exists_I Resclass$ always preferred to $\exists_I Sort$.

The essential subgoals of our proof are the lines L_{55} and L_{44}. Next the element of $(Int_4, \lambda xy \centerdot x \bar{*} y \bar{*} \bar{2}_4)$ is introduced whose trace has an order that is not reflected in the second structure. This is done in the usual way by applying $\exists_I Resclass$ and substituting the existentially quantified variable with the meta-variable mv_2. To restrict the search Multi gets hints for suitable instantiations for mv_1 and mv_2. The hints are computed by constructing the trace with Gap. In our example a suitable instantiation for mv_2 is 1 whose trace is $\{\bar{1}_4, \bar{2}_4, \bar{0}_4\}$. Thus, the corresponding suitable instantiation for mv_1 is 3 as the order of the trace. The trace itself is introduced by the application of the method *RewrTrace* to line L_{54}. The purpose of the method is to rewrite an *OrderTr*-statement into a corresponding statement on the cardinality of the set that constitutes the trace. *RewrTrace* is one of the domain specific methods the `TryAndError` strategy had to be extended with in order to deal with concepts of order, traces, cardinality, etc.

It is now left to show that the order of the traces of all elements in $(\mathbb{Z}_4, \lambda xy \centerdot \bar{2}_4)$ is different from 3, which is indeed the case as all traces are either of order 1 or 2. The proof is conducted with a regular case analysis on all elements of $\mathbb{Z}_4$ starting in line L_{44}. Here the search is again reduced by computing both traces and their order with Gap.

As opposed to the direct use of `TryAndError` the described combination of `ReduceToSpecial` and `TryAndError` has only polynomial complexity in the cardinality of the involved sets. Moreover, the search is reduced significantly by providing

L_1.	$L_1 \vdash Iso(\mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	(Hyp)
	$\vdots$	
L_6.	$L_1 \vdash Inj(h, \mathbb{Z}_4)$	$(\wedge_E^* \ldots)$
L_7.	$L_1 \vdash Hom(h, \mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(\wedge_E^* \ldots)$
L_8.	$L_1 \vdash h(\bar{0}_4) = \bar{2}_4$	$(InstHomEq\ L_7)$
	$\vdots$	
L_{13}.	$L_1 \vdash h(\bar{2}_4) = \bar{2}_4$	$(InstHomEq\ L_7)$
	$\vdots$	
L_{91}.	$L_1 \vdash \bar{2}_4 = \bar{2}_4$	$(=Reflexivity)$
L_{92}.	$L_1 \vdash h(\bar{0}_4) = \bar{2}_4$	$(=Subst\ L_{91}\ L_8)$
L_{93}.	$L_1 \vdash h(\bar{0}_4) = h(\bar{2}_4)$	$(=Subst\ L_{92}\ L_{13})$
	$\vdots$	
L_{97}.	$L_1 \vdash \neg Inj(h, \mathbb{Z}_4)$	$(\ldots)$
L_{98}.	$L_1 \vdash \bot$	$(\neg_E\ L_{97}\ L_6)$
L_{99}.	$\vdash \neg Iso(\mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$	$(Contra\ L_{98})$

Table 8.6: Proof with the `NotInjNotIso` strategy.

hints. But this technique is only applicable when structures involved contain elements suitable for our purpose in the sense that either their order or the order of their trace is not reflected in the respective other structure.

8.3.3 Proof by Contradiction

In this section we introduce a new strategy, `NotInjNotIso`, which is based on the idea to construct an indirect proof to show that two structures $(RS^1_{n_1}, \circ_1)$ and $(RS^2_{n_2}, \circ_2)$ are not isomorphic. We first assume that there exists a bijective function $h{:}RS^1_{n_1} \to RS^2_{n_2}$. If h is an isomorphism, then it is in particular an injective homomorphism. The strategy `NotInjNotIso` tries to find two elements $c_1, c_2 \in RS^1_{n_1}$ with $c_1 \neq c_2$ such that we can derive the equation $h(c_1) = h(c_2)$. This contradicts the assumption of injectivity of h where $h(c_1) \neq h(c_2)$ has to hold if $c_1 \neq c_2$. Note that the proof works with respect to all possible homomorphism h and we do not have to give a particular mapping.

Table 8.6 shows an extract of the proof with the `NotInjNotIso` strategy for our example problem $\neg Iso(\mathbb{Z}_4, \lambda xy_\bullet x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy_\bullet \bar{2}_4)$. The idea is to derive the contradiction in line L_{98} by assuming that there actually exists an isomorphism in line L_1. In particular, we use the properties that all possible isomorphisms h have to be injective homomorphisms given in lines L_6 and L_7. To line L_7 the domain specific method *InstHomEq* is applied which introduces the completely instantiated homomorphism equation system into the proof. In our example this system comprises 16 single equations. In figure 8.6 we give only two of these equations in lines L_8 and L_{13} to preserve space. The application of *InstHomEq* already introduces the simplified versions of equations, which are of the general form $h(x \circ_1 y) = h(x) \circ_2 h(y)$. When instantiating the proper operations and applying those to the arguments $x = \bar{0}_4$ and $y = \bar{0}_4$ we obtain the equation of line L_8 (similarly we receive line L_{13} from $x = \bar{1}_4$ and $y = \bar{1}_4$).

From the introduced system of equations the `NotInjNotIso` strategy tries to derive that h is not injective. To prove this we have to find two witnesses c_1 and c_2 such that $c_1 \neq c_2$ and $h(c_1) = h(c_2)$. In the proof in figure 8.6 we choose $\bar{0}_4$ and $\bar{2}_4$ for c_1 and c_2, respectively. We omit the part of the proof that derives $\bar{0}_4 \neq \bar{2}_4$

and rather concentrate on the more difficult part to show $h(\bar{0}_4) = h(\bar{2}_4)$ in line L_{93}. This goal is transformed into an equation that can be solved in a general way, by successively applying equations from the equation system. In our example $h(\bar{0}_4) = h(\bar{2}_4)$ is reduced in two steps to $\bar{2}_4 = \bar{2}_4$, which can be justified with the reflexivity of equality. Since line L_{97} contradicts the assumption of injectivity of h, MULTI can conclude the proof.

In order to restrict the search for appropriate c_1 and c_2 `NotInjNotIso` employs a control rule to obtain a hint. The control rule calls MAPLE to compute all possible solutions for the system of instantiated homomorphism equations with respect to the corresponding modulo factor using MAPLE's function `msolve`. Then the solutions are checked whether there is a pair c_1 and c_2 with $c_1 \neq c_2$, such that in every solution $h(c_1) = h(c_2)$ holds. If there is such a pair it is provided as hint. Although the control rule cannot always come up with a hint, our experiments have shown that the `NotInjNotIso` strategy is also often successful when no hint can be computed.

In our example the equational reasoning involved is still relative trivial and could be done by a more specialized system such as a term rewriting system. However, this is not possible in the general case. Then the equations contain more complex terms involving addition, multiplication, and subtraction of constant congruence classes of the form $h(cl_n(i))$ and thus additionally have to be performed with respect to the correct modulo factor. The solution of the equations is therefore beyond the scope of any term rewriting system but requires symbolic computation. Whereas in our example the equation in line L_{91} is justified by the reflexivity of the equality, in the general case more complicated equations are closed by applying the more general *SolveEqu* method, in which MAPLE is employed to solve the equation in question.

As in our example, `NotInjNotIso` can produce very short proofs even for structures with large sets. However, to construct an appropriate sequence of equality substitutions is generally the hard part of proofs with `NotInjNotIso`. In fact, for problems with the same complexity (i.e., problems involving structures of the same cardinality) the lengths of the proofs can vary drastically. Moreover, the equational reasoning process does not have to terminate. Therefore, we have an upper bound on the maximum number of equations to be tested before the strategy fails. This bound is currently 50. For these reasons, we are currently experimenting with randomization and restart techniques [100] to improve the strategies behavior.

`NotInjNotIso` is the first strategy that is tried when automatically discharging non-isomorphism proof obligations. If it fails our standard order of strategies take over; that is, since the `EquSolve` strategy is not applicable to non-isomorphism problems, `ReduceToSpecial` is tried before `TryAndError`.

8.4 Direct Products

With minor extensions to our strategies the proof techniques described in this section are also applicable to proofs where the structures involved contain direct products of residue class sets. Apart from those methods already illustrated in section 7.3.4 that decompose quantifications and equations on tuples into the components, a few additions had to be made for both isomorphism proofs and non-isomorphism proofs.

In the case of direct products in the domain or codomain of the mapping the pointwise defined function introduced by the `TryAndError` strategy in isomorphism proofs maps tuples of residue classes to tuples of meta variables. For example, in an isomorphism proof the pointwise function for the mapping h from $RS^1_{n_1} \otimes RS^2_{n_2}$

to $RS^3_{n_3} \otimes RS^4_{n_4}$, has the form

$$h(x,y) = \begin{cases} (mv_1, mv_2), \text{ if } (x,y) = (c_1,c_1) \in RS^1_{n_1} \otimes RS^2_{n_2} \\ (mv_3, mv_4), \text{ if } (x,y) = (c_1,c_2) \in RS^1_{n_1} \otimes RS^2_{n_2} \\ \quad \vdots \end{cases},$$

with $mv_1, mv_3, \ldots \in RS^3_{n_3}$ and $mv_2, mv_4, \ldots \in RS^4_{n_4}$. In non-isomorphism proofs the codomain of the mapping contains constants instead of meta-variables.

Similarly, the interpolating mapping for the pointwise isomorphism function between direct products is a tuple of multivariate polynomials. We have one polynomial for each component of the direct product in the codomain. The number of variables of each of these polynomials corresponds to the number of components of the direct product in the domain. For the example above, an interpolation for the function h is the pair $(P_1(x,y), P_2(x,y))$ consisting of two polynomials in two variables P_1 and P_2.

For the `NotInjNotIso` strategy we have one separate equation system for each component of the direct product in the codomain. Each equation system is of the form $h_i(x \circ_1 y) = h_i(x) \circ_2 h_i(y)$, with $1 \leq i \leq n$ and n is the number of components. Then we have to show for each equation system separately that $h_i(c_1) = h_i(c_2)$ with $c_1 \neq c_2$. Here x, y, c_1, c_2 are elements of the residue class structure in the domain of the mapping and can also be tuples.

In isomorphism proofs involving direct products the `ReduceToSpecial` strategy can apply a theorem stating that in order to prove that two direct products with the same number of components are isomorphic it is sufficient to establish isomorphisms between appropriately chosen single components. For instance, $(\mathbb{Z}_2 \otimes \mathbb{Z}_3, \lambda xy_{\bullet} x \bar{+} y \otimes \lambda xy_{\bullet} x \bar{+} y \bar{+} \bar{1}_3)$ and $(\mathbb{Z}_3 \otimes \mathbb{Z}_2, \lambda xy_{\bullet} x \bar{+} y \otimes \lambda xy_{\bullet} x \bar{+} y \bar{+} \bar{1}_2)$ are isomorphic since the first component of the one structure is isomorphic to the second component of the second structure and vice versa.

8.5 Classifying Isomorphic Structures

We shall now present how we classify sets of residue class structures into *equivalence classes of isomorphic structures.* Currently we determine the isomorphism classes only for residue class structures with one binary operation. The idea of the classification algorithm is to partition a set of residue class structures into disjunct classes of isomorphic structures. We assume that the given structures are all of the same algebraic category and have the same cardinality (i.e., we use the results of the classification process described in chapter 7.4).

The input of the classification algorithm is such a set of structures. In case we do not have a set of isomorphism classes yet, we construct an isomorphism class initially containing the first of the input structures. Otherwise we start the following classification cycle, which is repeated for each structure S in the input set:

1. Check whether there exists already an isomorphism class $\mathcal{C}$ such that S is isomorphic to the structures in $\mathcal{C}$. This is tested by checking successively for all present isomorphism classes whether one of its structures is isomorphic to S or not. Since the relation isomorphic is transitive it is sufficient to perform this check with only one structure S' in $\mathcal{C}$, respectively.

2. If we can prove that S is isomorphic to a structure S' of an isomorphism class $\mathcal{C}$ then S is added to $\mathcal{C}$.

3. If we can prove for each currently existing isomorphism class that S is not isomorphic to one of its structures, then we create a new isomorphism class initially containing S.

The test in step 1 is in turn performed in three steps: We first perform a computation whose result gives us the likely answer to the question whether the two structures S and S' are isomorphic or not. This computation consist of constructing a pointwise isomorphic mapping between the two structures, which is computed with the aid of solutions for the equation system that corresponds to the homomorphism mapping between S and S'. The solutions are computed using MAPLE; the actual computation is described in more detail in section 8.2.1. But, opposed to the classification described in chapter 7.4, we do not construct and discharge a proof obligation of each check. Instead we first conduct all possible checks and then construct proof obligations.

If we have found an S' to which S is supposedly isomorphic we construct this proof obligation. Otherwise we construct for each isomorphism class C a proof obligations that S is not isomorphic to a $S' \in C$. This way we postpone and even avoid superfluous non-isomorphism proofs. The proof obligations are then discharged by constructing a proof plan with MULTI. In case MULTI cannot prove the proof obligation suggested by MAPLE's result (e.g., if MAPLE's solutions are not sufficient to produce an isomorphic mapping even if one exists) the algorithm proceeds by constructing the negated proof obligation and passes it again to MULTI to discharge it. In case this attempt fails too, the algorithm signals an error.

8.6 Experiments

The proof techniques presented in this chapter mainly build on the strategies already constructed for the proofs of simple properties of the residue class structures as presented in chapter 7. To develop the additions to the `ReduceToSpecial`, `TryAndError`, and `EquSolve` to handle isomorphism proofs we used 15 examples and another 4 examples to build the `NotInjNotIso` strategy.

We applied the techniques of this chapter to the results of the classification process presented in the previous chapter. To accelerate the classification we excluded structures that were trivially not isomorphic to each other. Hence, we only examined structures, which are of the same algebraic category (e.g., monoids are only compared with other monoids and not with groups) and of the same cardinality. This avoided the construction of the tedious proofs for the trivial cases, which are easily constructed by the planner with the `ReduceToSpecial` strategy.

Among the structures classified were 8128 structures with the set $\mathbb{Z}_6$. Here, we found 4152 magmas, 73 abelian magmas, 1114 semi-groups, 1025 abelian semi-groups, 738 quasi-groups, 257 abelian quasi-groups, 50 abelian monoids, and 419 abelian groups. On these classes we started our isomorphism classification and discovered that the quasi-groups and the abelian quasi-groups each belong to two different classes, whereas all abelian monoids and all abelian groups are isomorphic. Furthermore, we had three non-isomorphic classes of abelian semi-groups, seven classes of semi-groups, and five classes of abelian magmas. We did not perform the classification for the non-abelian magmas. All the necessary isomorphism proofs were done with the `EquSolve` strategy, where `ReduceToSpecial` was applied to the injectivity subproblem. During the automatic classification 46 non-isomorphism proofs were constructed; in addition to the automatic classification process we did separate experiments with 200 non-isomorphism proofs on residue class structures of the same cardinality regardless of their previous classification. Here 80% of the proofs were done with the NotInjNotIso strategy and the remaining 20% with the combination of TryAndError and ReduceToSpecial.

Overall a considerable part of the problems have been proved with various usage of computer algebra. On the one hand the strategies EquSolve and NotInjNotIso ultimately rely directly on the usage of Maple. On the other hand even in the TryAndError strategy the hints computed by Maple and Gap used to provide suitable instantiations for witness terms reduce the search drastically. In particular, for larger residue class sets proofs by exhaustive case analysis and crude force search are hardly feasible.

8.7 Summary of Chapter 8

This chapter presented a case study on proving isomorphisms and non-isomorphism between residue class structures. For the proofs we could reuse the strategies implemented for the case study of chapter 7 with only minor extensions. However, we had to add sophisticated algorithms to compute useful hints with the computer algebra systems. We conducted a significant number of experiments that especially demonstrated that the more elaborate strategies are necessary for successfully solving the given problems. In particular, planning non-isomorphism proofs turned out to be challenging, since here exhaustive case analysis even with appropriate hints is not feasible for residue class structures with cardinality larger than 4. Here the NotInjNotIso strategy is a very promising approach, however the proofs can vary significantly in length even for problems of the same complexity. Ideas to solve this variance problem with randomization and restart techniques [100] are currently under investigation [147, 149].

Chapter 9

Conclusion and Outlook

Ω-Ants, the topic of this thesis, is a novel approach to a flexible and adaptive suggestion mechanism in interactive theorem proving and proof planning. We have employed it for the combination of reasoning techniques in theorem proving and their application to group theory and finite algebra.

Ω-Ants is a hierarchical blackboard architecture that consists of two layers of blackboards with individual concurrent knowledge sources. The lower layer searches for instantiations of command parameters within the actual proof state; the upper layer exploits this information to assemble a set of applicable commands characterizing the possible proof steps and presents them to the user. The architecture has also mechanisms to adapt its behavior with respect to the current proof context and the availability of system resources.

This architecture can be employed for several purposes:

1. To support the user in interactive theorem proving and proof planning to search for the next possible proof step in-between user interactions.
2. To further the cooperation or competition between various integrated automatic components such as automated theorem provers, computer algebra systems, and model generators.
3. As a tool for automated theorem proving by automating the command applications.
4. To perform knowledge base queries during proof search.

The case studies we presented are based on known theorems and probably all of the presented theorems have been shown by some automaton in one way or the other. However, this is the first attempt to build the heuristically guided search into one theorem prover based on proof planning. The automatic classification of residue classes is the first systematic exploration of this domain.

The Ω-Ants mechanism in its current state is solely centered around the blackboard architecture as well as the single proof object $\mathcal{PDS}$. The latter also provides the communication platform for the cooperation of various integrated automatic components. This centralized approach could be given up for a more distributed approach in which the single components behave more like real agents. This way it may be possible to form clusters of agents to encourage either local cooperation (e.g., one could think of a cooperation between a higher and a first order theorem prover which could offset their respective weaknesses) or local competition between single reasoning agents (e.g., several first order provers could concurrently solve problems). Then a prerequisite is the solution of the communication problem, which has already been the bottleneck in the current version of Ω-Ants where sometimes more time is spend on the translation and communication of proofs than on actual proof search. This problem has, for instance, also been pointed out by Denzinger and Fuchs [74]. Since a uniform communication format produces too much overhead specialized communication languages for clusters could be constructed which the participating agents would have to negotiate and agree upon. Cooperation between systems would also be enhanced if more available systems could produce and return explicit partial proofs. Moreover, heuristical functions to decide for several partial proofs, which one is the most promising would have to be developed.

Part IV

Appendix

Appendix A

Overview on Defined Concepts

This chapter contains an overview on all the defined concepts in alphabetical order, which occur in this thesis, and which are necessary for the formalization of the problems in the case studies of part III. The concepts necessary for the formalization of the agents in chapter 3.6 are therefore omitted.

$\subseteq$	$\equiv$	$\lambda T_{\alpha o}\blacksquare\lambda S_{\alpha o}\blacksquare\forall x\blacksquare T(x) \Rightarrow S(x)$	(5.19)
$\bar{-}$	$\equiv$	$\lambda r_{\nu o}\blacksquare\lambda s_{\nu o}\blacksquare\lambda z_{\nu}\blacksquare\exists x_{\nu}{:}r\blacksquare\exists y_{\nu}{:}s\blacksquare z = x - y$	(7.13)
$\bar{+}$	$\equiv$	$\lambda r_{\nu o}\blacksquare\lambda s_{\nu o}\blacksquare\lambda z_{\nu}\blacksquare\exists x_{\nu}{:}r\blacksquare\exists y_{\nu}{:}s\blacksquare z = x + y$	(7.11)
$\bar{*}$	$\equiv$	$\lambda r_{\nu o}\blacksquare\lambda s_{\nu o}\blacksquare\lambda z_{\nu}\blacksquare\exists x_{\nu}{:}r\blacksquare\exists y_{\nu}{:}s\blacksquare z = x * y$	(7.12)
$\otimes$	$\equiv$	$\lambda U_{\alpha o}\blacksquare\lambda V_{\beta o}\blacksquare\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\blacksquare [LProj(p) \in U] \land [RProj(p) \in V]$	(7.26)
$\times$	$\equiv$	$\lambda U_{\alpha o}\blacksquare\lambda V_{\beta o}\blacksquare\lambda \circ^1_{\alpha\alpha\alpha}\blacksquare\lambda \circ^2_{\beta\beta\beta}\blacksquare\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\blacksquare\lambda q_{(\alpha\beta)((\alpha\beta o)o)}\blacksquare Pair(LProj(p) \circ^1 LProj(q), RProj(p) \circ^2 RProj(q))$	(7.27)
$Assoc$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\forall b_{\beta}{:}G\blacksquare\forall c_{\beta}{:}G\blacksquare (a \circ (b \circ c)) = ((a \circ b) \circ c)$	(5.3)
$Closed$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\forall b_{\beta}{:}G\blacksquare G(a \circ b)$	(5.2)
$Commu$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\forall b_{\beta}{:}G\blacksquare[(a \circ b) = (b \circ a)]$	(5.9)
cl	$\equiv$	$\lambda n_{\nu}\blacksquare\lambda m_{\nu}\blacksquare\lambda x_{\nu}\blacksquare[\mathbb{Z}(x)] \land [(x\ mod\ n) = m]$	(7.1)
$Distrib$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\lambda \star_{\beta\beta\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\forall b_{\beta}{:}G\blacksquare\forall c_{\beta}{:}G\blacksquare [(a \star (b \circ c)) = ((a \star b) \circ (a \star c))] \land [((a \circ b) \star c) = ((a \circ c) \star (b \circ c))]$	(5.10)
$Divisors$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\forall b_{\beta}{:}G\blacksquare [\exists x_{\beta}{:}G\blacksquare(a \circ x) = b] \land [\exists y_{\beta}{:}G\blacksquare(y \circ a) = b]$	(5.8)
$Group$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare NonEmpty(G) \land Closed(G, \circ) \land Assoc(G, \circ) \land [\exists e_{\beta}{:}G\blacksquare Unit(G, \circ, e)] \land Inverse(G, \circ, StructUnit(G, \circ))$	(5.16)
Hom	$\equiv$	$\lambda h_{\alpha\beta}\blacksquare\lambda A_{\alpha o}\blacksquare\lambda \circ_{\alpha\alpha\alpha}\blacksquare\lambda B_{\beta o}\blacksquare\lambda \star_{\beta\beta\beta}\blacksquare \forall x_{\alpha}{:}A\blacksquare\forall y_{\alpha}{:}A\blacksquare h(x \circ y) = h(x) \star h(y)$	(6.1)
Im	$\equiv$	$\lambda f_{\alpha\beta}\blacksquare\lambda A_{\alpha o}\blacksquare\lambda y_{\beta}\blacksquare\exists x_{\alpha}{:}A\blacksquare y = f(x)$	(6.4)
Inj	$\equiv$	$\lambda f_{\alpha\beta}\blacksquare\lambda A_{\alpha o}\blacksquare\forall x_{\alpha}{:}A\blacksquare\forall y_{\alpha}{:}A\blacksquare f(x) = f(y) \Rightarrow x = y$	(6.2)
$Inverse$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\lambda e_{\beta}\blacksquare \forall a_{\beta}{:}G\blacksquare\exists x_{\beta}{:}G\blacksquare[(a \circ x) = e] \land [(x \circ a) = e]$	(5.5)
Iso	$\equiv$	$\lambda A_{\alpha o}\blacksquare\lambda \circ_{\alpha\alpha\alpha}\blacksquare\lambda B_{\beta o}\blacksquare\lambda \star_{\beta\beta\beta}\blacksquare\exists h{:}F(A,B)\blacksquare Inj(h, A) \land Surj(h, A, B) \land Hom(h, A, \circ, B, \star)$	(8.1)
$Kern$	$\equiv$	$\lambda f_{\alpha\beta}\blacksquare\lambda A_{\alpha o}\blacksquare\lambda y_{\beta}\blacksquare\lambda x_{\alpha}\blacksquare[x \in A] \land [f(x) = y]$	(6.5)
$LeftInverse$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\lambda e_{\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare\exists x_{\beta}{:}G\blacksquare(x \circ a) = e$	(5.7)
$LeftStructUnit$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\iota e_{\beta}\blacksquare LeftUnit(G, \circ, e)$	(5.15)
$LeftUnit$	$\equiv$	$\lambda G_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare\lambda e_{\beta}\blacksquare\forall a_{\beta}{:}G\blacksquare(e \circ a) = a$	(5.6)
$Loop$	$\equiv$	$\lambda L_{\beta o}\blacksquare\lambda \circ_{\beta\beta\beta}\blacksquare NonEmpty(L) \land Closed(L, \circ) \land$	(5.18)

		$\mathit{Divisors}(L,\circ) \wedge \exists e_\beta{:}_G{\blacksquare}\mathit{Unit}(G,\circ,e)$	
LProj	$\equiv$	$\lambda p_{(\alpha\beta o)o}{\blacksquare}\iota x_\alpha{\blacksquare}\exists y_\beta{\blacksquare}p = \mathit{Pair}(x,y)$	(7.21)
Magma	$\equiv$	$\lambda M_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\mathit{NonEmpty}(M) \wedge \mathit{Closed}(M,\circ)$	(5.11)
Monoid	$\equiv$	$\lambda M_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\mathit{NonEmpty}(M) \wedge \mathit{Closed}(M,\circ)\wedge$ $\mathit{Assoc}(M,\circ) \wedge [\exists e_\beta{:}_M{\blacksquare}\mathit{Unit}(M,\circ,e)]$	(5.13)
$\mathit{NonEmpty}$	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\exists a{\blacksquare}G(a)$	(5.1)
Pair	$\equiv$	$\lambda x_\alpha{\blacksquare}\lambda y_\beta{\blacksquare}\lambda g_{\alpha\beta o}{\blacksquare}g(x,y)$	(7.20)
$\mathit{Quasigroup}$	$\equiv$	$\lambda Q_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}$ $\mathit{NonEmpty}(Q) \wedge \mathit{Closed}(Q,\circ) \wedge \mathit{Divisors}(Q,\circ)$	(5.17)
Res	$\equiv$	$\lambda c_{\nu o}{\blacksquare}\lambda n_\nu{\blacksquare}\iota m_\nu{\blacksquare}\forall x_\nu{\blacksquare}[x \in c] \Rightarrow [x \mathit{ mod } n = m]$	(7.6)
$\mathit{RightInverse}$	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\lambda e_\beta{\blacksquare}\forall a_\beta{:}_G{\blacksquare}\exists x_\beta{:}_G{\blacksquare}(a \circ x) = e$	
$\mathit{RightStructUnit}$	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\iota e_\beta{\blacksquare}\mathit{RightUnit}(G,\circ,e)$	
$\mathit{RightUnit}$	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\lambda e_\beta{\blacksquare}\forall a_\beta{:}_G{\blacksquare}(a \circ e) = a$	
RProj	$\equiv$	$\lambda p_{(\alpha\beta o)o}{\blacksquare}\iota y_\beta{\blacksquare}\exists x_\alpha{\blacksquare}p = \mathit{Pair}(x,y)$	(7.22)
RS	$\equiv$	$\lambda n_\nu{\blacksquare}\lambda r_{\nu o}{\blacksquare}\exists m_\nu{:}_{\mathbb{N}}{\blacksquare}[r = cl_n(m)] \wedge [\mathit{NonEmpty}(cl_n(m))]$	(7.2)
$\mathit{Semigroup}$	$\equiv$	$\lambda S_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}$ $\mathit{NonEmpty}(S) \wedge \mathit{Closed}(S,\circ) \wedge \mathit{Assoc}(S,\circ)$	(5.12)
$\mathit{StructUnit}$	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\iota e_\beta{\blacksquare}\mathit{Unit}(G,\circ,e)$	(5.14)
$\mathit{SubGroup}$	$\equiv$	$\lambda U_{\beta o}{\blacksquare}\lambda \star_{\beta\beta\beta}{\blacksquare}\lambda G_{\beta o}{\blacksquare}\lambda \circ_{b\beta\beta\beta}{\blacksquare}$ $[\star = \circ] \wedge [U \subseteq G] \wedge [\mathit{Group}(U,\star)] \wedge [\mathit{Group}(G,\circ)]$	(5.20)
Surj	$\equiv$	$\lambda f_{\alpha\beta}{\blacksquare}\lambda A_{\alpha o}{\blacksquare}\lambda B_{\beta o}{\blacksquare}\forall x_\beta{:}_B{\blacksquare}\exists y_\alpha{:}_A{\blacksquare}f(y) = x$	(6.3)
Unit	$\equiv$	$\lambda G_{\beta o}{\blacksquare}\lambda \circ_{\beta\beta\beta}{\blacksquare}\lambda e_\beta{\blacksquare}\forall a_\beta{:}_G{\blacksquare}[(a \circ e) = a] \wedge [(e \circ a) = a]$	(5.4)

Appendix B

Overview of the Proved Theorems

This chapter contains a detailed summary of all theorems proved during the experiments in the case studies presented in chapters 7 and 8. Due to its size it is issued as a separate technical report [152].

Bibliography

[1] Andrew A. Adams. Theorem Proving in Support of Computer Algebra — DITLU: A Definite Integral Table Lookup. Submitted to the LMS Journal of Computation and Mathematics.

[2] Andrew A. Adams, Hanne Gottliebsen, Steve A. Linton, and Ursula Martin. VSDITLU: a Verifiable Symbolic Definite Integral Table Look-Up. In Ganzinger [92], pages 112–126.

[3] Peter B. Andrews. Resolution in Type Theory. *Journal of Symbolic Logic*, 36(3):414–432, 1971.

[4] Peter B. Andrews. General Models and Extensionality. *Journal of Symbolic Logic*, 37(2):395–397, 1972.

[5] Peter B. Andrews. General Models, Descriptions and Choice in Type Theory. *Journal of Symbolic Logic*, 37(2):385–394, 1972.

[6] Peter B. Andrews. Transforming Matings into Natural Deduction Proofs. In Wolfgang Bibel and Robert A. Kowalski, editors, *Proceedings of the 5th Conference on Automated Deduction (CADE–5)*, volume 87 of *LNCS*, pages 281–292, Les Arcs, France, June 7–9 1980. Springer Verlag, Berlin, Germany.

[7] Peter B. Andrews. *An Introduction To Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, San Diego, CA, USA, 1986.

[8] Peter B. Andrews. On Connections and Higher Order Logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

[9] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[10] Alessandro Armando and Clemens Ballarin. Maple's Evaluation Process as Constraint Contextual Rewriting. In Bernard Mourrain, editor, *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 32–37, London, Ontario, Canada, July 22–25 2001. ACM Press, Berkeley, CA, USA.

[11] Alessandro Armando and Daniele Zini. Interfacing Computer Algebra and Deduction Systems via the Logic Broker Architecture. In Kerber and Kohlhase [122], pages 49–64.

[12] Serge Autexier, Dieter Hutter, Bruno Langenstein, Heiko Mantel, Georg Rock, Axel Schairer, Werner Stephan, Roland Vogt, and Andreas Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special issue on Mechanized Theorem Proving for Technology*, 1998.

[13] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. System Description: inka 5.0 - A Logic Voyager. In Ganzinger [92], pages 207–211.

[14] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In Hsiang [115], pages 397–402.

[15] Franz Baader and Klaus Schulz, editors. *Proceedings of First International Workshop Frontiers of Combinning Systems (FROCOS'96)*, volume 3 of *Series on Applied Logic*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[16] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic Paramodulation and Superposition. In Kapur [120], pages 462–476.

[17] Clemens Ballarin, Karstern Homann, and Jacques Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In A. H. M. Levelt, editor, *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC-95)*, pages 150–157, Montreal, Canada, July 10–12 1995. ACM Press, Berkeley, CA, USA.

[18] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica — an Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[19] Peter Baumgartner and Uli Furbach. PROTEIN, a PROver with a Theory INterface. In Bundy [53], pages 769–773.

[20] Christoph Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving*. PhD thesis, Faculty of Technology, Saarland University, 1999.

[21] Christoph Benzmüller, Matthew Bishop, and Volker Sorge. Integrating TPS and ΩMEGA. *Journal of Universal Computer Science*, 5(3):188–207, March 1999. Special issue on Integration of Deduction System.

[22] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Erica Melis, Andreas Meier, Wolf Schaarschmidt, Jörg Siekmann, and Volker Sorge. ΩMega: Towards a Mathematical Assistant. In McCune [142], pages 252–255.

[23] Christoph Benzmüller, Mateja Jamnik, Manfred Kerber, and Volker Sorge. An Agent-oriented Approach to Reasoning. In Linton and Sebastiani [134].

[24] Christoph Benzmüller, Mateja Jamnik, Manfred Kerber, and Volker Sorge. Experiments with an Agent-Oriented Reasoning System. In Franz Baader, Gerhard Grewka, and Thomas Eiter, editors, *KI 2001: Advances in artificial intelligence : Joint German/Austrian Conference on AI*, volume 2174 of *LNAI*, pages 409–424, Vienna, Austria, September 19–21 2001. Springer Verlag, Berlin, Germany.

[25] Christoph Benzmüller and Michael Kohlhase. Extensional Higher-Order Resolution. In Kirchner and Kirchner [125], pages 56–71.

[26] Christoph Benzmüller and Michael Kohlhase. LEO – a Higher Order Theorem Prover. In Kirchner and Kirchner [125], pages 139–144.

[27] Christoph Benzmüller and Volker Sorge. A Blackboard Architecture for Guiding Interactive Proofs. In Fausto Giunchiglia, editor, *Artificial Intelligence: Methodology, Systems and Applications, Proceedings of the of the 8th International Conference AIMSA'98*, volume 1480 of *LNAI*, pages 102–114, Sozopol, Bulgaria, September 21–23 1998. Springer Verlag, Berlin, Germany.

[28] Christoph Benzmüller and Volker Sorge. Critical Agents Supporting Interactive Theorem Proving. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA-99)*, volume 1695 of *LNAI*, pages 208–221, Évora, Portugal, September 21–24 1999. Springer Verlag, Berlin, Germany.

[29] Christoph Benzmüller and Volker Sorge. Resource Adaptive Agents in Interactive Theorem Proving. SEKI-Report SR-99-02, Universität des Saarlandes, March 1999.

[30] Christoph Benzmüller and Volker Sorge. Towards Fine-Grained Proof Planning with Critical Agents. In Manfred Kerber, editor, *Informal Proceedings of the Sixth Workshop on Automated Reasoning Bridging the Gap between Theory and Practice in conjunction with AISB'99 Convention*, pages 20–22, Edinburgh, Scotland, April 8–9 1999. extended abstract.

[31] Christoph Benzmüller and Volker Sorge. Ω-ANTS – An open approach at combining Interactive and Automated Theorem Proving. In Kerber and Kohlhase [122], pages 81–97.

[32] Piergiorgio G. Bertoli, Jacques Calmet, Fausto Giunchiglia, and Karsten Homann. Specification and Integration of Theorem Provers and Computer Algebra Systems. In Jacques Calmet and Jan Plaza, editors, *Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation (AISC-98)*, volume 1476 of *LNAI*, pages 94–106. Springer Verlag, Berlin, Germany, September 16–18 1998.

[33] Piergiorgio G. Bertoli, Jacques Calmet, Fausto Giunchiglia, and Karsten Homann. Specification and Integration of Theorem Provers and Computer Algebra Systems. *Fundamenta Informaticae*, 39:39–57, 1999.

[34] Wolfgang Bibel and Peter H. Schmitt, editors. *Automated Deduction – A Basis for Applications*, volume 2. Kluwer, 1998.

[35] Matthew Bishop and Peter B. Andrews. Selectively Instantiating Definitions. In Kirchner and Kirchner [125], pages 365–380.

[36] Maria Paola Bonacina. Ten years of parallel theorem proving: a perspective (Invited paper). In Bernhard Gramlich, Hélène Kirchner, and Frank Pfenning, editors, *Proceedings of FLOC-99 Workshop on Stategies in Automated Deduction (STRATEGIES'99)*, pages 3–15, Trento, Italy, July 5 1999.

[37] Maria Paola Bonacina. A Taxonomy of Parallel Strategies for Deduction. *Annals of Mathematics and Artificial Intelligence*, in press, 2001.

[38] Maria Paola Bonacina and Jieh Hsiang. Parallelization of Deduction Strategies: an Analytical Study. *Journal of Automated Reasoning*, 13:1–33, 1994.

[39] Alan H. Bond and Les Gasser. An analysis of Problems and Research in DAI. In Alan H. Bond and Less Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 3–35. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1988.

[40] George Boole. *An Investigation of The Laws of Thought.* Macmillan, Barclay, & Macmillan, Cambridge, United Kingdom, 1854.

[41] Peter Borovansky, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN V 3.3 User manual. Technical report, INRIA Lorraine & LORIA, Nancy, France, December 1998.

[42] Sylvain Boulmé, Thérèse Hardin, Daniel Hirschkoff, Valérie Ménissier-Morain, and Renaud Rioboo. On the way to Certify Computer Algebra Systems. In Alessandro Armando and Tudor Jebelean, editors, *CALCULEMUS 99, Systems for Integrated Computation and Deduction*, volume 23(3) of *Electronic Notes in Theoretical Computer Science*, pages 51–66, Trento, Italy, July 11–12 1999. Elsevier. URL: `http://www.elsevier.nl/locate/entcs`.

[43] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Some Hints for Polynomials in the FOC Project. In Linton and Sebastiani [134], pages 142–154.

[44] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, USA, 1979.

[45] Robert S. Boyer and J Strother Moore. Metafunctions. In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, San Diego, CA, USA, 1981.

[46] Robert S. Boyer and J Strother Moore. *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic*, volume 11 of *Machine Intelligence.* Oxford University Press, Oxford, United Kingdom, 1988.

[47] Thomas Brüggemann, François Bry, Norbert Eisinger, Tim Geisler, Sven Panne, Heribert Schütz, Sunna Torge, and Adnan Yahya. Satchmo: Minimal Model Generation and Compilation. In Jean-Luis Imbert, editor, *Proceedings V-èmes Journées Francophones de Programmation en Logique et Programmation par Contraintes (JFPLC'96). Prototypes.*, pages 9–14, Clermont-Ferrand, France, June 1996. Hermes, Paris, France.

[48] Bruno Buchberger, Klaus Aigner, Claudio Dupre, Tudor Jebelean, Franz Kriftner, Mircea Marin, Ovidiu Podisor Koji Nakagawa, Elena Tomuta, Yaroslav Usenko, Daniela Vasaru, and Wolfgang Windsteiger. Theorema: An Integrated System for Computation and Deduction in Natural Style. In *Proceedings of the CADE-15 Workshop on Integration of Deductive Systems*, pages 96–102, Lindau, Germany, July 5–6 1998.

[49] Reinhard Bündgen. Application of the Knuth-Bendix Completion Algorithm to Finite Groups. Technical report, Univ. Tübingen, Tübingen, Germany, 1989.

[50] Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. Parallel ReDuX $\to$ PaReDuX. In Hsiang [115], pages 408–413.

[51] Reinhard Bündgen, Manfred Göbel, Wolfgang Küchlin, and Andreas Weber. Parallel Term Rewriting with PaReDuX. In Bibel and Schmitt [34], pages 231–260.

[52] Alan Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In Lusk and Overbeek [135], pages 111–120.

[53] Alan Bundy, editor. *Proceedings of the 12th International Conference on Automated Deduction (CADE–12)*, volume 814 of *LNAI*, Nancy, France, June 26–July 1 1994. Springer Verlag, Berlin, Germany.

[54] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction (CADE–10)*, volume 449 of *LNAI*, pages 647–648, Kaiserslautern, Germany, 1990.

[55] John Byrnes. *Proof Search and Normal Forms in Natural Deduction*. PhD thesis, Department of Philosophy, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1999.

[56] Jacques Calmet and Karsten Homann. Classification of Communication and Cooperation Mechanisms for Logial and Symbolic Computation Systems. In Baader and Schulz [15], pages 133–146.

[57] Jacques Calmet and Karsten Homann. Towards the Mathematical Software Bus. *Theoretical Computer Science*, 187(1–2):221–230, 1997.

[58] Jaques Calmet and Carla Limongelli, editors. *Design and Implementation of Symbolic Computation Systems; International Symposium, DISCO '96; Proceedings*, volume 1128 of *LNCS*, Karlsruhe, Germany, September 18–20 1996. Springer Verlag, Berlin, Germany.

[59] John Cannon and Catherine Playoust. *Algebraic Programming with Magma* Volume 1 and 2. Springer Verlag, Berlin, Germany, 1998. forthcoming.

[60] Lassaad Cheikhrouhou and Volker Sorge. $\mathcal{PDS}$ — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, March 22–24 2000.

[61] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. An introduction to Geometry EXpert. In McRobbie and Slaney [145], pages 235–239.

[62] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[63] Edmund Clarke and Xudong Zhao. Analytica-A Theorem Prover in Mathematica. In Kapur [120], pages 761–763.

[64] Dominique Clément, Francis Montagnac, and Vincent Prunet. Integrated Software Components: a Paradigm for Control Integration. In Albert Endres and Herbert Weber, editors, *Proceedings of the European Symposium on Software Development Environments and CASE Technology*, volume 509 of *LNCS*, pages 167–177, Königswinter, Germany, June 1991. Springer Verlag, Berlin, Germany.

[65] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W.Rance Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the* Nuprl *Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[66] Projet Coq. The Coq Proof Assistant (Version 7.0) — Reference Manual. Technical report, ENS Lyon - INRIA Rocquencourt, April 25 2001.

[67] Bernd Ingo Dahn, Jürgen Gehne, Thomas Honigmann, Lutz Walther, and Andreas Wolf. Integrating Logical Functions with ILF. Technical Report 94-10, Naturwissenschaftliche Fakultät II, Institut für Mathematik, Humboldt Universität zu Berlin, Germany, 1994.

[68] Bernd Ingo Dahn, Jürgen Gehne, Thomas Honigmann, and Andreas Wolf. Integration of Automated and Interactive Theorem Proving in ILF. In McCune [142], pages 57–60.

[69] Martin D. Davis and Hillary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):394–397, July 1960.

[70] Hans de Nivelle. *The Bliksem Theorem Prover, Version 1.12*. Max-Plank-Institut, Im Stadtwald, Saarbrücken, Germany, October 1999. Available from `http://www.mpi-sb.mpg.de/~bliksem/manual.ps`.

[71] Thomas Dean, editor. *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, July 31– August 6 1999. Morgan Kaufmann, San Mateo, CA, USA.

[72] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS-2000*, volume 1785 of *LNCS*, pages 78–92, Berlin, Germany, March 25–April 2 2000. Springer Verlag, Berlin, Germany.

[73] Jörg Denzinger and Bernd Ingo Dahn. Cooperating Theorem Provers. In Bibel and Schmitt [34], pages 483–416.

[74] Jörg Denzinger and Dirk Fuchs. Cooperation of Heterogeneous Provers. In Dean [71], pages 10–15.

[75] Jörg Denzinger and Martin Kronenburg. Planning for Distributed Theorem Proving: The Teamwork Approach. In Günther Görz and Steffen Hölldobler, editors, *Proceedings of the Twentieth Annual Conference on Artificial Intelligence (KI-96): Advances in Artificial Intelligence*, volume 1137 of *LNAI*, pages 43–56, Dresden, Germany, September 17–19 1996. Springer Verlag, Berlin, Germany.

[76] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT - A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.

[77] Peter Deussen. *Halbgruppen und Automaten*, volume 99 of *Heidelberger Taschenbücher, Sammlung Informatik*. Springer Verlag, Berlin, Germany, 1971.

[78] Mark Drummond. On Precondition Achievement and the Computational Economics of Automatic Planning. In Christer Bäckström and Erik Sandewall, editors, *Current Trends in AI Planning*, volume 20 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1994.

[79] R. Engelmore and T. Morgan, editors. *Blackboard Systems*. Addison-Wesley, 1988.

[80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and Raj Reddy. The HERSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 1980.

[81] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[82] Amy P. Felty and Douglas J. Howe. Hybrid Interactive Theorem Proving Using NUPRL and HOL. In McCune [142], pages 351–365.

[83] Armin Fiedler. Using a Cognitive Architecture to Plan Dialogs for the Adaptive Explanation of Proofs. In Dean [71], pages 358–363.

[84] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[85] Tim Finin, Rich Fritzson, Don McKay, and Robin McEntire. KQML - A Language and Protocol for Knowledge and Information Exchange. In *Proceedings of the 13th International Distributed AI Workshop*, pages 127–136, Seattle, WA, USA, 1994.

[86] Michael Fisher. An Open Approach to Concurrent Theorem Proving. In James Geller, Hiroaki Kitano, and Christian B. Suttner, editors, *Parallel Processing for Artificial Intelligence 3*, number 20 in Machine Intelligence and Pattern Recognition, pages 209–230. Elsevier/North Holland, Amsterdam, The Netherlands, 1997.

[87] Michael Fisher and Andrew Ireland. Multi-Agent Proof-Planning. In Jörg Denzinger, Michael Kohlhase, and Bruce Spencer, editors, *Proceedings of the CADE-15 Workshop on Using AI Methods in Deduction*, pages 33–42, Lindau, Germany, July 6 1998.

[88] Andreas Franke, Stephan M. Hess, Christoph G. Jung, Michael Kohlhase, and Volker Sorge. Agent-Oriented Integration of Distributed Mathematical Services. *Journal of Universal Computer Science*, 5(3):156–187, March 1999. Special issue on Integration of Deduction System.

[89] Gottlieb Frege. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle, Germany, 1879.

[90] Matthias Fuchs. The Application of Goal-Oriented Heuristics for Proving Equational Theorems via the Unfailing Knuth-Bendix Completion Procedure. A Case Study: Lattice Ordered Groups. SEKI-Report SR-94-02, Universität Kaiserslautern, Kaiserslautern, Germany, 1994.

[91] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic Generation of Some Results in Finite Algebra. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (ICJAI)*, pages 52–57, Chambery, France, August 28–September 3 1993. Morgan Kaufmann, San Mateo, CA, USA.

[92] Harald Ganzinger, editor. *Proceedings of the 16th International Conference on Automated Deduction (CADE–16)*, volume 1632 of *LNAI*, Trento, Italy, July 7–10, 1999. Springer Verlag, Berlin, Germany.

[93] Harald Ganzinger and Uwe Waldmann. Theorem Proving in Cancellative Abelian Monoids. In McRobbie and Slaney [145], pages 388–402.

[94] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4*, 1998. `http://www-gap.dcs.st-and.ac.uk/~gap`.

[95] Gerhard Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[96] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. Submitted, 2001.

[97] Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning Theories – Towards an Architecture for Open Mechanized Reasoning Systems. In Baader and Schulz [15], pages 157 – 174.

[98] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. English Version in [207].

[99] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1–2):67–100, February 2000.

[100] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search through Randomization. In Charles Rich and Jack Mostow, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence AAAI-98 and Tenth Conference on Innovative Application of Artificial Intelligence (IAAI-98)*, pages 431–437, Madison, WI, USA, July 26–30 1998. AAAI Press, Menlo Park, CA, USA.

[101] Mike J. Gordon, Arthur Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, Berlin, Germany, 1979.

[102] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, United Kingdom, 1993.

[103] Peter Graf. *Term Indexing*. Number 1053 in LNCS. Springer Verlag, Berlin, Germany, 1996.

[104] Marshall Hall. *The Theory of Groups*. The Macmillan Company, New York, NY, USA, 1959.

[105] John Harrison and Laurent Théry. Extending the HOL Theorem Prover with a Computer Algebra System to Reason About the Reals. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (HUG '93)*, volume 780 of *LNCS*, pages 174–184. Springer Verlag, Berlin, Germany, 1993.

[106] John Harrison and Laurent Théry. Reasoning About the Reals: The Marriage of HOL and Maple. In Andrei Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning LPAR'93*, volume 698 of *LNAI*, pages 351–353, St. Petersburg, Russia, July 1993. Springer Verlag, Berlin, Germany.

[107] John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

[108] Leon Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[109] Jaques Herbrand. *Recherches sur la théorie de la démonstration.* PhD thesis, Université de Paris, 1930. Englisch translation in [207].

[110] Thomas Hillenbrand, Andreas Jaeger, and Bernd Löchner. System Description: Waldmeister : Improvements in Performance and Ease of Use. In Ganzinger [92], pages 232–236.

[111] Karsten Homann. *Symbolisches Lösen mathematischer Probleme durch Kooperation algorithmischer und logischer Systeme.* PhD thesis, Unversität Karlsruhe, 1996. DISKI 152, Infix; St. Augustin.

[112] Karsten Homann and Jacques Calmet. An Open Environment for Doing Mathematics. In Michael Wester, Stanly Steinberg, and Michael Jahn, editors, *Proceedings of 1st International IMACS Conference on Applications of Computer Algebra*, Albuquerque, NM, USA, 1995.

[113] Karsten Homann and Jacques Calmet. Structures for Symbolic Mathematical Reasoning and Computation. In Calmet and Limongelli [58], pages 217–228.

[114] Douglas J. Howe. Computational Metatheory in NUPRL. In Lusk and Overbeek [135], pages 238–257.

[115] Jieh Hsiang, editor. *Proceedings of the* 6^{th} *International Consference on Rewriting Techniques and Applications*, volume 914 of *LNCS*, Kaiserslautern, Germany, April 5–7 1995. Springer Verlag, Berlin, Germany.

[116] Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach.* PhD thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, 1994.

[117] Xiaorong Huang. Reconstructing Proofs at the Assertion Level. In Bundy [53], pages 738–752.

[118] Malte Huebner. Resource adaptive theorem proving in Ω-ANTS. unpublished Class Project Report, 2000.

[119] Florian Kammüller and Lawrence C. Paulson. A Formal Proof of Sylow's First Theorem – An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, 23(3–4):235–264, November 1999.

[120] Deepak Kapur, editor. *Proceedings of the 11th International Conference on Automated Deduction (CADE–11)*, volume 607 of *LNAI*, Saratoga Spings, NY, USA, June 15–18 1992. Springer Verlag, Berlin, Germany.

[121] Matt Kaufmann and J Strother Moore. ACL2: An Industrial Strength Version of Nqthm. In *Eleventh Annual Conference on Computer Assurance (COMPASS '96)*, pages 23–34, Gaithersburg, Maryland, USA, June 17–21 1996. IEEE Computer Society Press.

[122] Manfred Kerber and Michael Kohlhase, editors. *Symbolic Computation and Automated Reasoning – The CALCULEMUS-2000 Symposium*, St. Andrews, United Kingdom, August 6–7, 2000 2001. AK Peters, Natick, MA, USA.

[123] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating Computer Algebra with Proof Planning. In Calmet and Limongelli [58], pages 204–215.

[124] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating Computer Algebra Into Proof Planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[125] Claude Kirchner and Hélène Kirchner, editors. *Proceedings of the 15th International Conference on Automated Deduction (CADE–15)*, volume 1421 of *LNAI*, Lindau, Germany, July 5–10 1998. Springer Verlag, Berlin, Germany.

[126] Hélène Kirchner and Christophe Ringeissen, editors. *Proceedings of Third International Workshop Frontiers of Combinning Systems (FROCOS 2000)*, volume 1794 of *LNCS*, Nancy, France, March 22–24 2000. Springer Verlag, Berlin, Germany.

[127] Lars Klein. Indexing für Terme höherer Stufe. Master's thesis, Computer Science Department, Universität des Saarlandes, 1997.

[128] Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[129] Michael Kohlhase. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle*. PhD thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, 1994.

[130] Robert Kowalski. A Proof Procedure Using Connection Graphs. *Journal of the Association for Computing Machinery (ACM), ACM, Inc., 1133 Avenue of the Americas, New York 10036*, 22(4):572–595, 1975.

[131] Ina Kraan, David Basin, and Alan Bundy. Middle-Out Reasoning for Logic Program Synthesis. Technical Report MPI-I-93-214, Max-Planck-Institut, Im Stadtwald, Saarbrücken, Germany, April 1993.

[132] Kenneth Kunen. Single Axioms for Groups. *Journal of Automated Reasoning*, 9(3):291–308, 1992.

[133] Bernhard Kutzler and Sabine Stifter. On the Application of Buchberger's Algorithm to Automated Geometry Theorem Proving. *Journal of Symbolic Computation*, 2(4):389–397, 1986.

[134] Steve Linton and Roberto Sebastiani, editors. *CALCULEMUS-2001 – 9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, Siena, Italy, June 21–22 2001.

[135] Ewing L. Lusk and Ross A. Overbeek, editors. *Proceedings of the 9th International Conference on Automated Deduction (CADE–9)*, volume 310 of *LNCS*, Argonne, Illinois, USA, 1988. Springer Verlag, Berlin, Germany.

[136] Ewing L. Lusk and Larry A. Wos. Benchmark Problems in Which Equality Plays the Major Role. In Kapur [120], pages 781–785.

[137] Ursula Martin. Theorem proving with group presentations: examples and questions. In McRobbie and Slaney [145], pages 388–402.

[138] John D. McCharen, Ross A. Overbeek, and Larry A. Wos. Problems and Experiments for and with Automated Theorem Proving Programs. *IEEE Transactions on Computers*, C-25(8):773–782, 1976.

[139] William McCune. Single Axioms for Groups and Abelian Groups with Various Operations. *Journal of Automated Reasoning*, 10(1):1–13, 1993.

[140] William McCune. A Davis-Putnam Program and Its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical Memorandum ANL/MCS-TM-194, Argonne National Laboratory, USA, 1994.

[141] William McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[142] William McCune, editor. *Proceedings of the 14th International Conference on Automated Deduction (CADE–14)*, volume 1249 of *LNAI*, Townsville, Australia, July 13–17 1997. Springer Verlag, Berlin, Germany.

[143] William McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, December 1997.

[144] William McCune and Ranganathan Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*, volume 1095 of *LNCS*. Springer Verlag, Berlin, Germany, 1996.

[145] Michael A. McRobbie and John K. Slaney, editors. *Proceedings of the 13th International Conference on Automated Deduction (CADE–13)*, volume 1104 of *LNAI*, New Brunswick, NJ, USA, July 30– August 3 1996. Springer Verlag, Berlin, Germany.

[146] Andreas Meier. Übersetzung automatisch erzeugter Beweise auf Faktenebene. Master's thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[147] Andreas Meier. Randomization and Heavy-Tailed Behavior in Proof Planning. Seki Report SR-00-03, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[148] Andreas Meier. Tramp: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, pages 460–464, Pittsburgh, PA, USA, June 17–20 2000. Springer Verlag, Berlin, Germany.

[149] Andreas Meier, Carla P. Gomes, and Erica Melis. Randomization and Restarts in Proof Planning. In Amedeo Cesta and Daniel Borrajo, editors, *Proceedings of the 6th European Conference on Planning (ECP-01)*, LNCS, Toledo, Spain, 2001. Springer Verlag, Berlin, Germany.

[150] Andreas Meier, Martin Pollet, and Volker Sorge. Exploring the Domain of Residue Classes. Seki Report SR-00-04, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, December 2000.

[151] Andreas Meier, Martin Pollet, and Volker Sorge. Classifying Isomorphic Residue Classes. In Roberto Moreno-Díaz, Bruno Buchberger, and José-Luis Freire, editors, *Proceedings of the 8th International Workshop on Computer Aided Systems Theory (EuroCAST 2001)*, volume 2178 of *LNCS*, pages 494–508, Las Palmas de Gran Canaria, Spain, February 19–23 2001. Springer Verlag, Berlin, Germany.

[152] Andreas Meier, Martin Pollet, and Volker Sorge. Classifying Residue Classes — Results of a Case Study. Seki Report SR-01-01, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, December 2001. Electronic Version at `http://www.ags.uni-sb.de/~sorge/publications/2001/SR-01-01.ps.gz`.

[153] Andreas Meier and Volker Sorge. Exploring Properties of Residue Classes. In Kerber and Kohlhase [122], pages 175–190.

[154] Erica Melis. The 'Interactive Textbook' Project. In Erica Melis, editor, *Proceedings of CADE-17 Workshop on Automated Deduction in Education*, pages 26–34, Pittsburgh, PA, USA, June 16 2000.

[155] Erica Melis and Andreas Meier. Proof Planning with Multiple Strategies. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic. 1st International Conference (CL2000)*, volume 1861 of *LNAI*, pages 644–659, London, United Kingdom, July 24–28 2000. Springer Verlag, Berlin, Germany.

[156] Erica Melis and Jörg Siekmann. Knowledge-Based Proof Planning. *Artificial Intelligence*, 115(1):65–105, November 1999.

[157] Dale Miller. *Proofs in Higher-Order Logic.* PhD thesis, Carnegie-Mellon University, 1983.

[158] Bernhard Hermann Neumann. Another Single Law for Groups. *Bulletin of the Australian Maths Society*, 23:81–102, 1981.

[159] Allen Newell, Cliff Shaw, and Herbert Simon. Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics. In *Proceedings of the 1957 Western Joint Computer Conference*, pages 218–239, New York, NY, USA, 1957. McGraw-Hill. Reprinted in Computers and Thought, Edward A. Feigenbaum, Julian Feldman, editors, New York, NY, USA, 1963, pages 109–133.

[160] H. Penny Nii, Nelleke Aiello, and James Rice. Frameworks for Concurrent Problem Solving: A Report on CAGE and POLIGON. In Engelmore and Morgan [79], pages 475–502.

[161] H. Penny Nii, Edward A. Feigenbaum, John J. Anton, and A.J. Rockmore. Signal-to-Symbol Transformation: HASP/SIAP Case Study. *AI Magazine*, 3(2):23–35, 1982.

[162] Lewis M. Norton. *ADEPT – A heuristic program for proving theorems of group theory.* PhD thesis, Massachusets Institute of Technology, 1966. Supervisor: Marwin Minsky.

[163] Hyacinth S. Nwana and Divine T. Ndumu. A Brief Introduction to Software Agent Technology. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, chapter 2, pages 29–47. Springer Verlag, Heidelberg, Germany, 1998.

[164] Jens Otten and Wolfgang Bibel. leanCoP: Lean Connection-Based Theorem Proving. In Peter Baumgartner and Hantao Zhang, editors, *Third International Workshop on First-Order Theorem Proving*, volume 5/2000 of *Research Report*, pages 152–157, St Andrews, United Kingdom, July 3–5 2000. Universität Koblenz-Landau, Germany.

[165] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer aided verification (CAV-96): 8th international conference*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, USA, July 31–August 3 1996. Springer Verlag, Berlin, Germany.

[166] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *LNCS*. Springer Verlag, Berlin, Germany, 1994.

[167] Francis Jeffry Pelletier. Seventy-Five Graduated Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[168] Frank Pfenning. *Proof Transformations in Higher-Order Logic.* PhD thesis, Carnegie-Mellon University, Pittsburgh Pa., 1987.

[169] Josef Pieprzyk and Babak Sadeghiyan. *Design of Hashing Algorithms*, volume 756 of *LNCS*. Springer Verlag, Berlin, Germany, 1993.

[170] Erik Poll and Simon Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor. In Kirchner and Ringeissen [126], pages 136–150.

[171] Axel Präcklein. The MKRP User Manual. Technical report, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[172] Dag Prawitz. *Natural Deduction – A Proof-Theoretical Study.* Acta Universitatis Stockholmiensis 3. Almqvist & Wiksell, Stockholm, Sweden, 1965.

[173] Zhenyu Qian. Unification of Higher-Order Patterns in Linear Time and Space. *Journal of Logic and Computation*, 6(3):315–341, 1996.

[174] Zhenyu Qian and Kang Wang. Modular AC Unification of Higher-Order Patterns. In Jean-Pierre Jouannaud, editor, *Constraints in Computational Logics, First International Conference, CCL'94*, volume 845 of *LNCS*, pages 105–120, Munich, Germany, September 7–9 1994. Springer Verlag, Berlin, Germany.

[175] Zhenyu Qian and Kang Wang. Modular Higher-Order Equational Preunification. *Journal of Symbolic Computation*, 22(4):401–424, 1996.

[176] I.V. Ramakrishnan, R. Sekar, and Andrei Voronkov. Term Indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1853–1964. Elsevier Science and MIT Press, Cambridge, MA, USA, 2001.

[177] Darren Redfern. *The Maple Handbook: Maple V Release 5.* Springer Verlag, Berlin, Germany, 1999.

[178] James Rice. The ELINT Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System. In N.S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 212–220, Detroit, MI, USA, August 20–25 1989. Morgan Kaufmann, San Mateo, CA, USA.

[179] Julian D.C. Richardson, Alan Smaill, and Ian M. Green. System description: Proof planning in higher-order logic with *λclam*. In Kirchner and Kirchner [125], pages 129–133.

[180] George A. Robinson and Larry Wos. Paramodulation and Theorem Proving in First Order Theories with Equality. In Bernhard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 133–150. American Elsevier, New York, USA, 1969.

[181] John Alan Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[182] Piotr Rudnicki. An Overview of the MIZAR Project. In *Proceedings of the 1992 Workshop on Types and Proofs as Programs*, pages 291–310, Båstad, 1992.

[183] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNAI*. Springer Verlag, Berlin, Germany, 1989.

[184] Johann Schumann and Ortrun Ibens. *SETHEO V3.3 Reference Manual (Draft)*. Institut für Informatik, TU München, 1997.

[185] Christian Schwarzweller. *Mizar Verification of Generic Algebraic Algorithms*. PhD thesis, Wilhelm-Schickard-Institute for Computer Science, University of Tuebingen, Germany, 1997.

[186] Wilfried Sieg and John Byrnes. Normal Natural Deduction Proofs (in classical logic). *Studia Logica*, 60(1):67–106, January 1998.

[187] Jon Siegel. *Corba: Fundamentals and Programming*. John Wiley & Sons Inc., Chichester, NY, USA, 1996.

[188] Jörg H. Siekmann and Graham Wrightson. Paramodulated Connection Graphs. *Acta Informatica*, 13:67–86, 1980.

[189] John Slaney. FINDER (Finite Domain Enumerator): Notes and guide. Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, Canberra, 1992.

[190] John Slaney, Masayuki Fujita, and Mark E. Stickel. Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems. *Computers and Mathematics with Applications*, 29:115–132, 1995.

[191] Konrad Slind, Mike Gordon, Richard Boulton, and Alan Bundy. An Interface between CLAM and HOL. In Kirchner and Kirchner [125], pages 129–133.

[192] Raymond M. Smullyan. *First-Order Logic*. Springer Verlag, Berlin, Germany, 1968.

[193] Volker Sorge. Integration eines Computeralgebrasystems in eine logische Beweisumgebung. Master's thesis, Universität des Saarlandes, November 1996.

[194] Volker Sorge. Non-Trivial Symbolic Computations in Proof Planning. In Kirchner and Ringeissen [126], pages 121–135.

[195] Jürgen Stuber. Superposition Theorem Proving for Abelian Groups Represented as Integer Modules. *Theoretical Computer Science*, 208(1–2):149–177, 1998.

[196] Patrick Suppes and Shuzo Takahashi. An Interactive Calculus Theorem-prover for Continuity Properties. *Journal of Symbolic Computation*, 7(1):573–590, January 1989.

[197] Geoff Sutcliffe and Christian B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[198] Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis. The TPTP Problem Library. In Bundy [53], pages 252–266.

[199] Christian B. Suttner and Johann Schumann. Parallel Automated Theorem Proving. In Laveen N. Kanal, Vipin Kumar, Hiroaki Kitano, and Christian B. Suttner, editors, *Parallel Processing for Artificial Intelligence 1*, number 14 in Machine Intelligence and Pattern Recognition, pages 209–257. Elsevier/North Holland, Amsterdam, The Netherlands, 1994.

[200] Moto-o Takahashi. Cut-Elimination in Simple Type Theory with Extensionality. *Journal of the Mathematical Society of Japan*, 19, 1968.

[201] Moto-o Takahashi. A system of simple type theory of Gentzen style with inference on extensionality and the cut-elimination in it. *Commentarii Mathematici Universitatis Sancti Pauli*, XVIII(II):129–147, 1970.

[202] Laurent Théry. A Certified Version of Buchberger's Algorithm. In Kirchner and Kirchner [125], pages 349–364.

[203] Simon Thompson. *Type Theory and Functional Programming.* International computer science series. Addison Wesley, Reading, MA, USA, 1991.

[204] Simon Thompson. Logic and dependent types in the Aldor Computer Algebra System. In Kerber and Kohlhase [122], pages 205–220.

[205] Andrzej Trybulec and Howard Blair. Computer Assisted Reasoning with MIZAR. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 26–28, Los Angeles, CA, USA, August 18–23 1985. Morgan Kaufmann, San Mateo, CA, USA.

[206] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the* AUTOMATH *System*, volume 83 of *Mathematical Centre Tracts.* Mathematisch Centrum, Amsterdam, The Netherlands, 1979.

[207] Jean van Heijenoort. *From Frege to Gödel : a source book in mathematical logic 1879-1931.* Source books in the history of the sciences series. Harvard Univ. Press, Cambridge, MA, 3rd printing, 1997 edition, 1967.

[208] Dongming Wang. GEOTHER: a geometry theorem prover. In McRobbie and Slaney [145], pages 166–170.

[209] Christoph Weidenbach, Bernd Gaede, and Georg Rock. SPASS & FLOTTER, version 0.42. In McRobbie and Slaney [145], pages 141–145.

[210] Gerhard Weiss, editor. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1999.

[211] Gerhard Weiss. Prologue. In Weiss [210], pages 1–23.

[212] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, volume I. Cambridge University Press, Cambridge, Great Britain; second edition, 1910.

[213] Andreas Wolf. P-SETHEO: Strategy Parallelism in Automated Theorem Proving. In Harrie de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98)*, volume 1397 of *LNAI*, page 320, Oisterwijk, The Netherlands, May 5–8 1998. Springer Verlag, Berlin, Germany.

[214] Michael J. Wooldridge. Intelligent Agents. In Weiss [210], pages 27–77.

[215] Larry A. Wos, George A. Robinson, and Daniel F. Carson. The Automatic Generation of Proofs in the Languange of Mathematics. In Wayne A. Kalenich, editor, *Information Processing 1965. Proceedings if IFIP Congress 65: Vol. 1 & 2*, volume 2, pages 325–326, New York City, USA, May 24–29 1965. Spartan [and others], Washington, D.C., USA.

[216] Wen-tsün Wu. *Mechanical Theorem Proving in Geometries : Basic Principles.* Texts and monographs in symbolic computation. Springer Verlag, Berlin, Germany, 1994.

[217] Hantao Zhang. SATO: An Efficient Propositional Prover. In McCune [142], pages 272–275.

[218] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[219] A.K. Zherlov and V.I. Martýanov. Automated Theorem Proving in Group Theory. *Algorithmic Topics in Algebraic Systems and Computers, University of Irkurtsk*, pages 36–64, 1979.

[220] Shlomo Zilberstein. Models of Bounded Rationality. In *AAAI Fall Symposium on Rational Agency*, Cambridge, Massachusetts, USA, November 1995.

[221] Zeljko Zilic and Katarzyna Radecka. On Feasible Multivariate Polynomial Interpolations over Arbitrary Fields. In Sam Dooley, editor, *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation (ISSAC-99)*, pages 67–74, Vancouver, BC, Canada, July 29–31 1999. ACM Press, Berkeley, CA, USA.

[222] Jürgen Zimmer. Constraintlösen für Beweisplanung. Master's thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany, May 2000.

[223] Richard Zippel. Probabilistic Algorithms for Sparse Polynominals. In Edward W. Ng, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (EUROSAM '79)*, volume 72 of *LNCS*, pages 216–226, Marseille, France, June 1979. Springer Verlag, Berlin, Germany.

[224] Richard Zippel. Interpolating Polynomials from Their Values. *Journal of Symbolic Computation*, 9(3):375–403, 1990.

List of Figures

List of Tables

Table of Defined Symbols

$\mathcal{T}_B$	set of base-types	13
$\mathcal{T}$	set of types	13
τ	type function	14
Σ	signature	14
$\neg$	negation	14
$\vee$	disjunction	14
Π^{α}	quantifier	14
ι^{α}	description operator	14
$\mathcal{V}$	set of variables	14
$\mathbf{wff}_{\alpha}(\Sigma)$	set of well formed formulas of α	14
$\mathbf{wff}(\Sigma)$	set of well-formed formulas over Σ	15
‘$\mathbf{.}$’	λ separator	15
$\mathbf{FV}(\mathbf{A})$	set of free variables of $\mathbf{A}$	15
$\rightarrow_{\alpha}$	α-conversion	15
$\rightarrow_{\beta}$	β-reduction	15
$\rightarrow_{\eta}$	η-reduction	15
$\mathbb{N}$	set of non-negative integers	15
$\mathbb{N}^*$	set of all words over $\mathbb{N}$	15
ϵ	empty word	15
‘.’	concatenation of words	15
$\langle\pi\rangle$	term position π	15
$\top$	truth	16
$\bot$	falsehood	16
$\mathcal{I}$	interpretation of constants	16
$\forall$	universal quantifier	16
$\exists$	existential quantifier	16

$\wedge$	conjunction	16
$\Rightarrow$	implication	16
$\Leftrightarrow$	equivalence	16
$\dot{=}^{\alpha}$	equality	16
φ	variable assignment	17
$\mathcal{I}_{\varphi}$	denotation	17
$\mathbf{M} = \langle \mathcal{D}, \mathcal{I} \rangle$	Henkin model	17
$\models$	semantical consequence	17
$\mathcal{H} \vdash_{ND} F$	syntactical consequence	20
$\mathcal{H} \vdash F$	syntactical consequence	20
$\equiv$	definition symbol	23
$\ominus$	delete list indicator	28
$\oplus$	add list indicator	28
$\mathcal{P}$	partial proof	41
$\mathcal{N}_{\mathcal{P}}$	set of proof nodes	41
ϵ	empty actual argument	41
$\mathcal{INST}$	set of possible actual arguments	41
$\mathfrak{G}$	goal agent	44
$\mathfrak{S}$	support agent	44
$\mathfrak{F}$	function agent	44
$\mathfrak{C}$	command agent	49
$\mathcal{P}$	partial proof	51
$<_n$	chronological node order	51
$\mathcal{SN}$	set of support nodes	51
$F_{\mathcal{P}}$	set of all foci	51
$FC_{\mathcal{P}}$	set of focus contexts	52
fc	focus context	52
$\prec$	foci priority order	52
pc	proof context	52
if-then-else	conditional branching	69
$cl_n(m)$	congruence class m modulo n	137
RS	residue class set	137
$\bar{+}$	addition on congruence classes	138
$\bar{*}$	multiplication on congruence classes	138
$\bar{-}$	subtraction on congruence classes	138

Index of Names

Index