Proof Planning with Multiple Strategies

Andreas Meier

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Saarbrücken, 2004

Dekan	Prof. Dr. Philipp Slusallek, Universität des Saarlandes, Saarbrücken
Vorsitzender	Prof. Dr. Raimund Seidel, Universität des Saarlandes, Saarbrücken
1. Gutachter	PD. Dr. Erica Melis, Universität des Saarlandes, Saarbrücken
2. Gutachter	Prof. Dr. Gert Smolka, Universität des Saarlandes, Saarbrücken
3. Gutachter	Prof. Dr. Jörg Siekmann, Universität des Saarlandes, Saarbrücken
Kolloquium	06.02.2004, Universität des Saarlandes, Saarbrücken

Abstract

This thesis presents proof planning with multiple strategies. Strategies are independent proof plan operations, and different strategies realize different plan refinements as well as plan modifications. Compared with the previous proof planning, multiple strategy proof planning introduces another hierarchical level and its heuristic control. Both, the strategies and the strategic control can encode (mathematical) domain knowledge.

We implemented proof planning with multiple strategies in the MULTI system. The evaluation of proof planning with multiple strategies and its implementation in MULTI is conducted with two large and two smaller case studies that are discussed in this thesis. The case studies illustrate the importance of domain knowledge at the strategy-level for proof planning.

Kurzzusammenfassung

Diese Arbeit stellt Beweisplanen mit mehreren Strategien vor. Strategien sind unabhängige Komponenten für das Beweisplanen, wobei verschiedene Strategien verschiedene Verfeinerungen oder Modifikationen eines Beweisplans realisieren können. Im Vergleich mit dem bisherigen Beweisplanen führt Beweisplanen mit mehreren Strategien eine neue Hierarchieebene und deren heuristische Kontrolle ein. Sowohl die Strategien selbst als auch ihre Kontrolle können (mathematisches) Wissen über eine Domäne kodieren.

Beweisplanen mit mehreren Strategien ist implementiert im MULTI System. Zur Evaluierung von Beweisplanen mit mehreren Strategien wurden mit MULTI zwei große und zwei kleinere Fallstudien durchgeführt, die in dieser Arbeit diskutiert werden. Die Fallstudien veranschaulichen das Domänenwissen, das auf der Ebene von Strategien vorliegt, und wie es im Beweisplanen benutzt werden kann.

Extended Abstract

Mathematicians prove complex theorems of a certain mathematical domain by flexibly combining several global and local problem solving strategies. In contrast, most of today's automated theorem proving systems use one or few strategies and typically their control is hard-coded into the systems algorithms. This was also true for ΩMEGA's previous proof planner, which combined the application of planning operators, the instantiation of variables, and backtracking in a pre-defined way. Moreover, the functionalities of these subcomponents were very restricted. The hard-coded combination of operations with restricted functionalities prohibited the use of mathematical knowledge of certain proof constructions and their combination. As a result, the planner failed on problems for which more flexibility and knowledge is needed in the proof planning process.

These observations led us to develop proof planning with multiple strategies, which we introduce in this thesis. The main idea is to decompose the previous monolithic proof planning process and to replace it by separate but collaborating operations, so-called strategies, which can realize different plan refinements and modifications. Moreover, the decision on when to apply a strategy should not be encoded once and forever into a fixed control procedure but rather be determined by meta-level reasoning using heuristic control knowledge of strategies and their combination. As compared with the previous proof planning, strategies and their heuristic control introduce another hierarchical level and can encode further (mathematical) domain knowledge.

The decomposition of the previous monolithic proof planner allows to extend and generalize the functionalities of its subcomponents. This results in independent parameterized algorithms for operator application, variable instantiation and backtracking. Technically, a strategy is an instantiation of such a parameterized algorithm and determines a certain behavior of the algorithm. The knowledge encoded into strategies can be diverse. Strategies can describe, for instance, different techniques to prove a class of problems. Strategies can also describe different ways of backtracking or different ways of constructing mathematical objects to instantiate variables.

Although the initial motivation for proof planning with multiple strategies was the decomposition of the previous monolithic proof planning process the new framework is open for the integration of all kinds of algorithms and their strategies that can contribute to a theorem proving process. Further algorithms integrated so far are an algorithm for the expansion of complex steps, an algorithm for cased-based reasoning, and an algorithm for the application of automated theorem provers.

To enable the flexible combination of individual strategies, multiple-strategy proof planning allows for meta-reasoning about the applicable strategies with declaratively stated heuristic control knowledge. Heuristic control knowledge is encoded into so-called strategic control rules, which can reason about the proof plan constructed so far, the plan process history, and the mathematical domain of the proof

planning problem. When evaluated with respect to a set of applicable strategies, strategic control rules can prefer promising strategies or can reject strategies whose application will likely result in a failed proof attempt. For instance, strategic control rules can guide the change of strategies during the proof planning process to tackle different subproblems with different strategies. Strategic control rules can delay or promote instantiations of variables, if this is heuristically preferable with respect to the current proof planning process. Strategic control rules can also handle failures during the proof planning process, for instance, when none of the available planning operators is applicable or when variables cannot be instantiated. In multi-strategy proof planning such a failure does not necessarily cause backtracking as in the previous proof planner of ΩMEGA. Rather, since failures often hold the key for the discovery of a solution proof plan, a strategic control rule can analyze the failure and can use it productively by suggesting particular plan refinements or modifications.

We implemented proof planning with multiple strategies in the MULTI system. MULTI has a blackboard architecture. We decided for a blackboard architecture since blackboard systems do not rely on a pre-defined control of the application of their components but provide the flexibility to employ the components in an event-driven way.

The evaluation of multiple-strategy proof planning and its implementation in MULTI is conducted with two large case studies and two smaller case studies from different mathematical domains that are discussed in this thesis. The case studies illustrate the importance of domain knowledge at the strategy-level for proof planning. In particular, we discuss example problems that cannot be solved with the previous monolithic proof planner of ΩMEGA since their solution requires the flexible combination of different proof plan refinements. MULTI can solve these problems and also all problems provable with the previous proof planner.

Ausführliche Zusammenfassung

Mathematiker beweisen Sätze in einem konkreten mathematischen Gebiet, indem sie eine Vielzahl von lokalen und globalen Lösungsstrategien flexibel kombinieren. Im Gegensatz dazu verfügen die meisten heutigen automatischen Beweiser nur über eine sehr eingeschränkte Menge von Strategien, welche zudem meist nicht flexibel kombinierbar sind. Typischerweise ist ein bestimmter Kontrollfluß in das System einprogrammiert. Dies galt auch für den bisherigen Beweisplaner des ΩMEGA Systems, dessen Kombination von Operationen wie etwa Anwendung eines Planungsoperators, Instantiierung einer Variablen und Backtracking fest einprogrammiert waren. Außerdem konnte viel vorhandenes mathematisches Wissen über Beweisplanverfeinerungen und -modifikationen nicht in den alten Beweisplaner eingebracht werden. Dies führte dazu, dass der Planer solche Beweisprobleme nicht lösen konnte, für die ein flexiblerer Planungsprozess nötig ist.

Diese Beobachtungen motivierten die Entwicklung von Beweisplanen mit mehreren Strategien, das wir in dieser Arbeit vorstellen. Die grundlegende Idee ist, den bisherigen Beweisplanungsprozess, in dem alle Teilkomponenten fest integriert sind, zu zerlegen und durch unabhängige Komponenten, sogenannte Strategien, zu ersetzen, die verschiedene Planverfeinerungen und -modifikationen realisieren können. Desweiteren sollte die Entscheidung, wann eine Strategie angewandt wird, nicht mehr in einem festen Kontrollzyklus vorgegeben werden, sondern sollte flexibel getroffen werden durch die Benutzung von heuristischem Kontrollwissen über Strategien und ihre Kombination. Verglichen mit dem bisherigen Beweisplanen führen Strategien und ihre heuristische Kontrolle eine neue Hierarchieebene ein und erlauben weiteres (mathematisches) Domänenwissens zu kodieren.

Die Zerlegung des bisherigen Planungsprozesses und die dadurch auch ermöglichte Erweiterung der Funktionalitäten seiner Teilkomponenten liefern unabhängige parametrisierte Algorithmen für Operator Anwendung, Variablen Instantiierung und Backtracking. Eine Strategie ist dann eine Instantiierung eines solchen parametrisierten Algorithmus und legt ein bestimmtes Verhalten des Algorithmus fest. Das in Strategien kodierte Wissen kann sehr vielfältig sein. So können Strategien zum Beispiel beschreiben, wie eine Klasse von Problemen auf verschiedene Art und Weise gelöst werden kann, Strategien können verschiedene Arten von Backtracking realisieren oder sie können verschiedene Möglichkeiten zur Konstruktion mathematischer Objekte zum Instantiieren von Variablen beschreiben.

Die ursprüngliche Motivation für Beweisplanen mit mehreren Strategien war, die Operationen des bisherigen Beweisplaners zu zerlegen. Der neu entwickelte Ansatz ist aber prinzipiell offen für die Integration beliebiger Algorithmen und deren Strategien, die zum Beweisprozess beitragen können. Beispielsweise wurden bisher ein Algorithmus zur Expansion komplexer Schritte, ein Algorithmus zum Beweisen mittels Analogie sowie ein Algorithmus für die Anwendung automatischer Beweiser integriert.

Um zwischen anwendbaren Strategien abwägen zu können und die flexible Kombination einzelner Strategien zu ermöglichen benutzt Beweisplanen mit mehreren Strategien deklaratives heuristisches Kontrollwissen. Heuristisches Kontrollwissen wird in sogenannten strategischen Kontrollregeln kodiert, die vorhandene Information über den momentanen Beweisplan, den bisherigen Beweisplanprozess und die mathematische Domäne des Problems auswerten. Die strategischen Kontrollregeln bevorzugen dann die Anwendung vielversprechender Strategien und verhindern die Anwendung von Strategien, die wahrscheinlich nicht zu einer Lösung führen würden. Zum Beispiel können strategische Kontrollregeln den Wechsel von Strategien während eines Planungsprozesses steuern, um verschiedene Teilprobleme mit verschiedenen Strategien anzugehen, die für das jeweilige Teilproblem geeignet scheinen. Strategische Kontrollregeln können auch die Instantiierung von Variablen vorziehen oder verzögern, je nachdem, ob die Instantiierung im momentanen Planungszustand heuristisch sinnvoll erscheint oder nicht. Andere strategische Kontrollregeln behandeln Fehler, die während des Beweisplanprozesses auftreten, z.B. wenn keine verfügbaren Planungsoperatoren anwendbar sind oder wenn Variablen nicht instantiiert werden können. Im Gegensatz zum vorherigen Beweisplaner von ΩMEGA ziehen Fehler beim Beweisplanen mit mehreren Strategien nicht notwendigerweise Backtracking nach sich. Vielmehr können strategische Kontrollregeln Fehler analysieren und darauf aufbauend bestimmte Planverfeinerungen oder -modifikationen steuern. Denn manchmal enthalten auftretende Fehler den Schlüssel zum Finden einer Lösung.

Wir haben Beweisplanen mit mehreren Strategien in dem neuen System MULTI implementiert. MULTI hat eine Blackboardarchitektur, die es erlaubt, Strategien bedarfsorientiert und durch die Auswertung von strategischen Kontrollregeln aufzurufen.

Zur Evaluierung von Beweisplanen mit mehreren Strategien und seiner Implementierung in MULTI wurden zwei große und zwei kleinere Fallstudien aus verschiedenen mathematischen Domänen durchgeführt, die in dieser Arbeit diskutiert werden. Die Fallstudien veranschaulichen das Domänenwissen, das auf der Ebene von Strategien vorliegt, und wie es im Beweisplanen benutzt werden kann. Insbesondere werden in der Arbeit Probleme diskutiert, die vom bisherigen Planer von ΩMEGA nicht gelöst werden konnten, da ihre Lösung die flexible Kombination verschiedener Planverfeinerungen benötigt. MULTI kann diese Probleme lösen sowie auch alle Probleme, die bereits der alte Planer lösen konnte.

Acknowledgments

Foremost, I want to thank my tutor Erica Melis for her collaboration and support. With many fruitful discussions she helped me to achieve a deeper insight into proof planning and the fundamental parts of my work. Moreover, I owe her a great debt of gratitude for carefully proof reading drafts of this thesis.

I am also greatly indebted to Jörg Siekmann, who introduced me into his ΩMEGA group in Saarbrücken. This not only gave me the opportunity to pursue my research, but the ΩMEGA group also provided a very stimulating environment that contributed to the success of my work.

I am grateful to Gert Smolka for accepting to be the second referee of this thesis.

For many fruitful discussions and collaborations in the ΩMEGA group I want to thank Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Moreover, my thanks is also to the following colleagues of the AG Siekmann outside the ΩMEGA group: Serge Autexier, Dieter Hutter, and Axel Schairer. For help with the implementation I am also indebted to Siegfried Scholl. For proof reading single parts of my thesis I thank Christoph Benzmüller, Manfred Kerber, and Claus-Peter Wirth.

Furthermore, I am grateful to Carla Gomes and Manfred Kerber for enabling me to visit and work with their respective research groups at Cornell University in Ithaca, NY, USA and at the University of Birmingham, UK. I also want to thank the Deutscher Akademischer Austauschdienst for funding my stay in Ithaca and the EU for funding my stay in Birmingham.

Above all, I want to thank my wife Susanne for her support and patience during the whole time of my thesis and especially for putting up with all my moods during the final months of my work. Finally, I am deeply grateful to my parents for their support over the years.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem

Typically, human experts have different problem solving techniques at their disposal that they can flexibly employ when solving a complex problem, for instance, when discovering a complex proof for a mathematical theorem. In particular, the choice of appropriate problem solving approaches are crucial human skills and are typically guided by some meta-reasoning.

For automated theorem proving the situation is quite different currently. Traditional logic-oriented automated theorem provers such as OTTER or SPASS search for proofs in the huge search spaces that result from the use of low-level logic rules. To traverse the search space they use search heuristics determined by parameter settings. These search heuristics are general-purpose heuristics such as the set-of-support technique or ordering techniques that hardly cover mathematical proof discovery heuristics. Moreover, it is not possible to change the search strategy during a proof attempt in order to adapt to the needs of subproblems. Thus, these systems cannot combine different search strategies.

An alternative technique for automated theorem proving is proof planning introduced by BUNDY in 1988. Proof planning considers a theorem to be proved as an Artificial Intelligence (AI) planning problem. BUNDY's key idea was to augment tactics that originate from tactical theorem proving with pre- and postconditions that specify the applicability of the tactic as well as its effects with respect to a proof state. This results in planning operators, so-called methods, which are more abstract than logic calculus rules. A proof planner searches for a sequence of method applications that derives a theorem from given assumptions, so that the automated proof search is performed at the abstract level of methods.

Another important advantage of proof planning is the possibility to incorporate domain-specific mathematical knowledge into the planning process. This was realized in the knowledge-based proof planning of the ΩMEGA system, which is developed by SIEKMANN and his group since the mid 1990s.

The previous proof planner of ΩMEGA provides two ways to encode knowledge, methods and control rules. ΩMEGA's methods can encode general proof steps as well as steps particular to a mathematical domain. Heuristic conditions about the desirability of the application of methods are encoded in control rules. Control rules allow, in particular, to encode global search control that can cover mathematical control knowledge. The control rules guide the search for a solution plan by preferring promising search paths or pruning search paths that are likely to lead to no solution. The previous proof planner performs a fixed cycle of method selection and application that is guided by control rules. This cycle is combined in a fixed way with restricted facilities for backtracking and for the instantiation of variables.

The application of this previous proof planner of ΩMEGA to problems from different mathematical domains revealed the following drawbacks. First, its subcomponents for method application, backtracking, and variable instantiation have only restricted functionalities that do not enable, for instance, different kinds of backtracking or the realization of different ways to instantiate variables. Second, the integration of these subcomponents is hard-coded into the algorithm, so that they cannot be flexibly combined. As a consequence, this planner realizes only one particular hard-coded problem solving approach, which is suitable for many problems but insufficient for other ones. In particular, there is no possibility to adapt it to the needs of different classes of problems since large parts of its control are hard-coded.

Another problem with the previous proof planner originates from the fact that mathematics is knowledge-intensive. Hence, the exploration of different mathematical domains results in large sets of methods and control rules. This large amount of available knowledge can be used only, if it is appropriately structured into computationally manageable and conceptually sensible units. The previous proof planner of ΩMEGA, however, provides no means to structure sets of methods and control rules.

During our experiments with the previous proof planner of ΩMEGA we found knowledge about several proof plan refinements and modifications that are useful in certain situations. We also learnt how to combine these refinements and modifications. For instance, we discovered sets of methods and control rules that belong together since they encode together the knowledge of how to tackle a certain class of problems (i.e., they encode together a certain proof technique to prove problems from the class). We found that the instantiation of variables should be flexibly combined with the introduction of methods since in some situations it is useful to delay the instantiation of variables whereas in other situations it is useful to promote the instantiation. By analyzing failed proof attempts we learnt about different useful kinds of backtracking. In other situations the failures themselves hold the key to discover a solution. Hence, the analysis of such a failure gives rise to the suggestion of particular proof plan refinements or modifications rather than to backtrack. All this knowledge of proof plan refinements and modifications and their controlled combination cannot be represented in methods and control rules. Hence, there is no means to incorporate and use it in the previous proof planner of ΩMEGA.

1.2 Solutions

To overcome the problems of knowledge-based proof planning that originate from the rigidity of the hard-coded problem solving approach of the previous monolithic proof planner (as discussed in the previous section) this thesis presents proof planning with multiple strategies. This novel approach is implemented in a new proof planner called MULTI.

The main idea of proof planning with multiple strategies is to decompose the previous monolithic proof planning process and to replace it by separate but collaborating operations, so-called strategies, which can realize different plan refinements and modifications. Moreover, the decision on when to call a strategy should not be encoded once and forever into the system but rather be determined by meta-level reasoning using heuristic control knowledge of strategies and their combination. As compared with the previous proof planning, strategies and their heuristic control introduce another hierarchical level and can encode further (mathematical) domain knowledge.

Algorithms and Strategies

The decomposition of the previous monolithic proof planner of ΩMEGA allows to extend and generalize the functionalities of its subcomponents. This results in independent parameterized algorithms for method application, variable instantiation and backtracking. A strategy is an instantiation of such a parameterized algorithm and determines a certain behavior of the algorithm. When a strategy is invoked, then its algorithm is applied to the current proof planning state with respect to the parameter instantiation specified by the strategy.

The multiple-strategy proof planning framework is not restricted to the algorithms resulting from the decomposition of the previous proof planner. Rather, it is open for the integration of all kinds of algorithms and their strategies that can contribute to a theorem proving process. Currently, MULTI employs the following 6 independent and parameterized algorithms:

PPLANNER refines a proof plan by introducing new method steps.

INSTMETA refines a proof plan by instantiating variables.

BACKTRACK modifies a proof plan by removing refinements of other algorithms.

EXP refines a proof plan by expanding complex steps.

ATP refines a proof plan by solving subproblems with traditional automated theorem provers.

CPLANNER refines a proof plan by transferring steps from a source proof plan or fragment.

PPLANNER, **INSTMETA**, and **BACKTRACK** result from the decomposition and generalization of the subcomponents of the previous proof planner of ΩMEGA. **EXP**, **ATP**, and **CPLANNER** integrate new refinements of the proof plan.[1]

The knowledge encoded into strategies can be diverse. For instance, the algorithm **PPLANNER** has parameters for a set of methods and a set of control rules. Thus, a **PPLANNER** strategy specifies a set of methods and control rules, for instance, methods and control rules that encode together a proof technique to prove a certain class of problems. Several **PPLANNER** strategies provide a means to structure the available method and control rule knowledge into units of methods and control rules that belong together. Strategies of **INSTMETA** determine different ways to construct mathematical objects to instantiate variables, for instance, by employing different kinds of external systems to provide instantiations for variables. Strategies of **BACKTRACK** determine different ways to backtrack by deleting different sets of steps.

Strategic Control

Knowledge of the applicability of strategies is subdivided into knowledge of the legal feasibility of a strategy and knowledge of the heuristic desirability of strategies. The

[1] **CPLANNER** adapts and extends functionalities of the TOPAL system, a component of ΩMEGA for cased-based reasoning.

legal conditions that have to be satisfied in order for a strategy to be applicable are part of the specification of the strategy. Heuristic knowledge about the desirability of certain strategies in particular situations is encoded into strategic control rules, which guide the search at the strategy-level similar to control rules at the method-level. Strategic control rules can reason about the proof plan constructed so far, the plan process history, and the mathematical domain of the proof planning problem. When evaluated with respect to a set of applicable strategies, strategic control rules can prefer promising strategies or can reject strategies whose application will likely result in a failed proof attempt.

The advantage of this declarative and knowledge-based control approach is that the heuristic control of proof planning with multiple strategies can be easily extended and changed by modifying the strategic control rules. In contrast, when the combination of integrated components of a system is hard-coded into a control procedure, then each extension or change requires re-implementation of parts of the main control procedure. Moreover, declaratively stated control knowledge can be communicated more easily to a user in order to clarify and explain taken decisions. However, the acquisition and implementation of suitable control knowledge can be difficult, but it is typically necessary for the successful application of proof planning.

Similar to the knowledge in strategies also the knowledge encoded in strategic control rules can be diverse. For instance, strategic control rules can guide the switch of **PPLANNER** strategies during the proof planning process to tackle different subproblems with different sets of methods and control rules that seem to be appropriate for the respective subproblem. Strategic control rules can also guide the combination between **PPLANNER** strategies and the strategies of other algorithms. For instance, strategic control rules can delay or promote instantiations of variables performed by strategies of **INSTMETA**, if this is heuristically preferable with respect to the current proof planning process. Strategic control rules can also handle failures during the proof planning process, for instance, when none of the available planning operators is applicable or when variables cannot be instantiated. In multi-strategy proof planning such a failure does not necessarily cause backtracking as in the previous proof planner of ΩMEGA. Rather, since failures often hold the key for the discovery of a solution proof plan, a strategic control rule can analyze the failure and can use it productively by suggesting particular plan refinements or modifications.

Implementation in MULTI

For the implementation of the multiple-strategy proof planning approach in MULTI we decided for a blackboard architecture since blackboard architectures have proven useful to organize the cooperation of several independent components, so-called knowledge sources, for solving a complex problem. This is because blackboard systems do not rely on a pre-defined control of the application of the involved components but employ their knowledge sources event-driven, i.e., whenever possible and suitable. MULTI's architecture consists of two blackboards, the so-called proof blackboard and the control blackboard. The two-blackboard architecture emphasizes the importance of both, the solution of the proof planning problem whose status is stored on the proof blackboard and the solution of the control problem, that is, which possible strategy should the system apply next. Corresponding to the two blackboards, there are two sets of knowledge sources that work on these blackboards: the strategies work on the proof blackboard whereas the knowledge source that works on the control blackboard is called the MetaReasoner. It evaluates the strategic control rules in order to guide the selection of the next strategy.

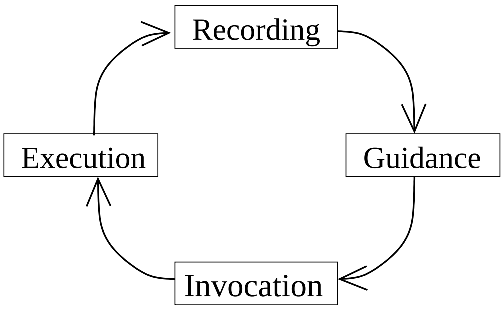


Figure 1.1: Control cycle of Multi.

In a nutshell, Multi operates according to the control cycle in Figure 1.1, which passes the following steps:

Recording Strategies whose condition is true record their applicability on the control blackboard.

Guidance The MetaReasoner evaluates the strategic control rules to order the applicability records on the control blackboard.

Invocation A scheduler invokes the strategy who posed the highest ranked applicability record.

Execution The algorithm of the invoked strategy is executed with respect to the parameter instantiation specified by the strategy.

Except for this cycle, no control is hard-coded into Multi. In particular, no preference or exclusion of strategies is pre-defined. There are several strategic control rules that define a 'reasonable' default control for Multi. For instance, there is a strategic control rule that rejects strategies that failed already. Another rule suggests backtracking, if a failure occurs. Although these control rules are the backbone of Multi's control, they can be excluded by the user of Multi or can be overridden by other strategic control rules. For instance, in the case studies conducted with Multi, we developed more specific control rules that allow for the repeated application of the same strategy although it failed already. Moreover, we developed more specific strategic control rules that analyze and productively use failures to suggest particular plan refinements or modifications rather than to backtracking.

1.3 Case Studies

For an evaluation of multiple-strategy proof planning and its implementation in Multi we present two large case studies and two smaller case studies that we conducted with Multi. They show that multiple-strategy proof planning naturally extends simple proof planning and extends the problem solving horizon of proof planning.

1. The first case study investigates proof planning for theorems taken from the analysis textbook [12] about the limit of sequences, the limit of functions, the continuity of functions, and the derivative of functions. This domain was first tackled with Ωmega's previous proof planner. The analysis of the failed attempts of the previous proof planner strongly influenced the design of Multi.

The case study demonstrates how proof planning with multiple strategies enables the flexible integration of a constraint solver to provide instantiations for variables and reasoning about failures to guide backtracking and the subsequent proof planning process. For instance, failures can be exploited to guide the eureka steps of lemma speculation and case-split introduction.

In this case study we discuss, in particular, example problems that cannot be solved with the previous proof planner of ΩMEGA since their solution requires the flexible instantiation of variables and the flexible handling of failures. MULTI can solve these problems (as well as all other problems provable with the previous proof planner) since it can make use of the additional domain knowledge encoded into strategies and strategic control rules.

2. The second case study is concerned with the automatic classification of residue class structures with respect to their algebraic properties and with respect to isomorphic structures. To solve problems from this domain we realized several proof techniques in several proof planning strategies. The availability of several proof techniques for one problem makes proof planning more robust: if one proof technique fails on a problem, another proof technique may solve it. The case study also benefits from different kinds of backtracking in MULTI and their guidance by reasoning about failures. Moreover, the case study demonstrates how MULTI supports the flexible integration of computer algebra systems, model generators, theory formation systems, and automated theorem provers with proof planning.

3. In the third case study, we apply MULTI to solve problems on permutation groups. Essential for the success of MULTI in this domain are the incorporation of a computer algebra system and the hierarchical construction of proof plans. That is, proof planning in this domain exploits, among others, MULTI's algorithm for the expansion of complex steps and combines it with the other proof plan refinements and modifications.

4. The fourth case study applies MULTI to homomorphism problems. Although MULTI can solve the homomorphism problems automatically the main focus of the case study is to tackle these problems interactively with MULTI. The case study demonstrates how also interactive proof planning benefits from the new approach.

1.4 Overview

This thesis consists of three parts. Part I introduces the preliminaries of the thesis, part II describes MULTI, and part III contains descriptions of the case studies. The single parts are organized as follows:

Part I: Preliminaries After brief overviews of theorem proving with computers, blackboard systems, and Artificial Intelligence planning in chapter 2, we introduce the ΩMEGA system in chapter 3 and formally describe its underlying logic and its proof objects. In chapter 4, we shall introduce the basics of knowledge-based proof planning. In addition to technical descriptions of methods and control rules we shall give a formal definition of proof plans and a detailed description of the previous proof planner of ΩMEGA. We conclude part I in chapter 5 with a brief discussion of the theorems that are part of the limit domain and the residue class domain, since these problems are used throughout the rest of the thesis as examples.

Part II: Multi This part consists of two chapters. Chapter 6 introduces proof planning with multiple strategies and Multi. It starts with a motivation of the development of proof planning with multiple strategies. Then, it introduces the basic elements of proof planning with multiple strategies as well as Multi's blackboard architecture. It concludes with a discussion of the realized approach and a comparison with related work. Chapter 7 gives a technical description of Multi and the algorithms it employs so far.

Part III: Case Studies The case studies are described in three chapters. Chapters 8 and 9 describe the application of Multi to the limit domain and the residue class domain, respectively. The subject of chapter 10 is then the application of Multi to problems of permutation groups and homomorphism theorems.

Finally, chapter 11 concludes the thesis with a summary and an outlook to possible extensions.

Part I

Preliminaries

Chapter 2

Background

In this chapter we give a brief overview of the background of this thesis, namely theorem proving with computers, blackboard systems, and Artificial Intelligence planning.

2.1 Theorem Proving with Computers

Theorem proving systems were among the earliest Artificial Intelligence (AI) systems in the 1950s. For instance, at the Dartmouth Conference in 1956 DAVIS decision procedure based on Presburger's Arithmetic [66] and NEWELL and SIMON's Logic Theorist [178] were among the presented systems. Since this time a large variety of systems and approaches to automate and mechanize mathematical reasoning has been developed. We categorize these approaches into three classes: machine-oriented automated theorem proving, logic-oriented interactive theorem proving, and mathematics-oriented theorem proving.

2.1.1 Machine-Oriented Theorem Proving

> *It seems as though logicians had worked with the fiction of man as a persistent and unimaginative beast who can only follow rules blindly, and then the fiction found its incarnation in the machine.*
>
> Wang, 1960, quoted from [216], p. 260

Machine-oriented theorem provers are automated theorem provers (ATPs) based upon computational logical inference system such as *resolution* [205], *tableaux* [221], or *connection calculi* [142]. These systems search for a sequence of low-level logic rule applications that proves a theorem from a given set of axioms. The search is guided by general-purpose heuristics such as the set-of-support technique or ordering techniques [146] that hardly cover mathematical proof discovery heuristics. The strength of the systems stems from their ability to traverse and maintain very large search spaces (up to millions of nodes).

The breakthrough for machine-oriented theorem provers came with the work of WANG [238] and the development of the *resolution principle* by ROBINSON in 1965 [205]. Today many such theorem provers exist for different logics. For propositional logic there are, for instance, SAT-based systems such as SATO [251] and ANL-DP [149], which rely on the *Davis-Putnam Procedure* [67]. For first-order logic a myriad of systems has been developed. Representatives of systems that are based on the resolution principle are MKRP [197], Otter [150], Bliksem [68], and Spass [240]. SETHEO [212] is a prover based on the tableaux calculus and leanCoP [187] uses a connection calculus. For higher-order logic there are systems based on the (suitably extended) resolution principle such as the Leo system [19] and systems based on the (suitably extended) connection method such as tps [8].

For specific classes of problems there are also specialized systems. For instance, an important subfield of automated theorem proving are so-called *term rewriting systems*. Term rewriting systems have been developed to prove whether an equality can be derived from a given set of input equations. A well known approach from this subfield is the *Knuth-Bendix completion* [138]. Representatives for term rewriting systems are WaldMeister [114] and eqp [152].

Like other applications of computers, machine-oriented theorem provers did profit from the development of faster computers with more memory. Due to this technological progress and due to the development of very efficient implementation techniques (e.g., sophisticated indexing techniques [108, 199]) machine-oriented provers have been successfully applied in logic and mathematics (e.g., see [250]) and succeeded to prove non-trivial open mathematical problems such as the Robbins Algebra Conjecture [152].

Nevertheless, machine-oriented theorem provers suffer from the explosion of the search space that results from their low-level inference systems. Consequently, many problems of well-understood mathematical domains are beyond the capabilities of today's systems. The mathematical knowledge and experience that humans employ to accomplish proofs in these domains cannot be used by the machine-oriented provers in their low-level search with logic inference rules. An example of such a domain are theorems about the limit of functions. In 1990 Bledsoe proposed several versions of the theorem that the limit of the sum of two functions over the reals equals the sum of their limits as a challenge problem for automated theorem proving [28]. Only the simplest versions of this problem (problem 1 and 2 in [28]) can be solved by today's machine-oriented automated theorem provers. The more difficult versions as well as theorems such as that the limit of the product of two functions over the reals equals the product of their limits are beyond their capabilities.

2.1.2 Logic-Oriented Interactive Theorem Proving

> *Some workers in automatic theorem proving, including the authors, believe it will be many years (if ever) before machines alone can prove difficult theorems in mathematics. Thus some, who hope to see machines used as practical assistants to pure mathematicians, have redirected their attention to man-machine theorem provers and theorem proof checking.*
>
> Bledsoe and Bruell, 1973,[26]

Despite the early enthusiasm for machine-oriented automated theorem provers it turned out that their applications in the daily work of a mathematician were limited. First, these provers fail often on main-stream mathematical problems; second, their output format is incomprehensible for humans; and third, essentially they work as a blackbox and give either a perfect answer (i.e., a proof) or no answer at all. This motivated the development of interactive systems to assist mathematicians by constructing and checking their proofs.

Although there were approaches to use variations of resolution as principle means to interactively construct proofs (e.g., see [2, 119]) most interactive systems are based on *natural deduction* [96] or *sequent calculi* [198], which are considered to be more human-oriented than resolution, tableaux, or connection calculi.

One of the earliest interactive provers of this paradigm is the AUTOMATH system developed by DE BRUIJN in the early 1970s [232]. AUTOMATH and other early systems suffered from the problem that proofs in their underlying natural deduction or sequent calculi have to be derived at a very fine-grained level, which requires many user interactions and results in very long proof objects (when compared to proofs in mathematical texts).

More recent interactive systems such as NUPRL [3], ISABELLE [189], HOL [107], and PVS [188] use tactics for proof construction. The idea in tactical theorem proving is that repeatedly occurring sequences of inference steps are encapsulated into macro steps, so-called tactics. Most tactic-based theorem proving systems (e.g., NUPRL, ISABELLE, HOL) are descendants of LCF [106] and follow a bottom-up approach for tactic construction. That is, more and more complex tactics are constructed by the decomposition of inference rules of the basic calculus and already defined tactics. Since such a tactic eventually results in the application of calculus level rules, a tactic may fail to be applicable, but if it is applicable, then it does not produce faulty steps.

The invention of tactics facilitated the use of interactive systems for proof construction and proof checking, and a large set of proofs has been constructed with these systems for mathematical applications (e.g., see [62]) as well as for program and hardware verifications (e.g., see [55, 143]). However, these approaches have not reached a broad acceptance as a working instrument for mathematicians. They may result in new standards of rigor in mathematical proofs but they focus on the logical correctness of steps and proofs, rather than to focus on the integration of mathematical knowledge and practice into the proof development process.

2.1.3 Mathematics-Oriented Theorem Proving

> *Automated theorem proving* [...] *is not the beautiful process we know as mathematics. This is 'cover your eyes with blinders and hunt through a cornfield for a diamond-shaped grain of corn'. Mathematicians have given us a great deal of direction over the last two or three millennia. Let us pay attention to it.*
>
> Bledsoe, 1986,[27]

Although the field of automated and interactive theorem proving with computers has been dominated by logic-oriented systems there have always been approaches that try to base theorem proving on mathematical knowledge and practice. Examples for such systems are GELERNTER's Geometry-Theorem Proving Machine [94] for Euclidean geometry theorems, BUNDY's SUMS prover [37] for part of arithmetic, and BLEDSOE's IMPLY [29] prover[1] for limit theorems.

The Geometry-Theorem Proving Machine was motivated by the fact that humans typically first draw a diagram to have a model of the problem at hand when proving a theorem of Euclidean geometry. This is because, *"the creative scientist generally finds his most valuable insights into a problem by considering a model of the formal system in which the problem is couched"* (quoted from [95], p. 103). Technically, the Geometry-Theorem Proving Machine uses two representations of the problem during the theorem proving process: a 'syntax-machine' constructs a proof of the given problem with rules and axioms on Euclidean geometry and a 'diagram-machine' maintains and updates a diagram, i.e., a model, of the problem

[1] To be more precise, the actual program was called PROVER and IMPLY was its principal subroutine for accomplishing limit theorems, see [29] for details.

(the initial diagram is given by the user).[2] The 'syntax-machine' backwardly applies rules and axioms to reduce the initial theorem to new subgoals. The 'diagram-machine' guides this proof search by rejecting those applications that result in subgoals that are false in the diagram and by instantiating variables in new subgoals, such that the subgoals are true in the diagram.

SUMS proves arithmetic theorems by representing them in the form of a diagram. The nodes of the diagram are property lists of arithmetic terms and its links describe relationships such as $<, \leq, =$. Knowledge about arithmetic is built into the system in form of procedures that draw the diagram, so that when links are added to it, elementary deductions are made (and more links are added) automatically without the explicit use of axioms of arithmetic or explicit inference rules. This results in a kind of proof protocol rather than a formal logic proof. However, the main intention of SUMS was not to produce formal proofs but to simulate the behavior of mathematicians as BUNDY points out: *"Does* SUMS *prove theorems or does it check their validity? It certainly does not produce proofs in a formal logical system* [...] *Nor, of course, does the practicing mathematician confine himself to either of these techniques. Rather he is prepared to use a variety of methods to achieve his ends. To convince himself, and others, he produces a protocol. Formal logical systems were introduced to analyze and justify this procedure and not to replace it as a method of discovery.* SUMS *is designed to simulate the behavior of mathematicians. During the course of a proof it 'proves' many facts (i.e., convinces itself of their truth) and records these as true; it also produces a protocol which is intended to convince others of their truth (i.e., a proof)."* (quoted from [37])

Limit theorems turned out to be a difficult domain for machine-oriented automated theorem provers since they require the axioms of an ordered field that cause long and difficult searches. Motivated by the fact that *"a human mathematician is often able to easily perform the necessary operations of analysis without being aware of the explicit use of the field axioms"* (quoted from [29], p. 586) IMPLY employs knowledge on the limit domain in form of routines for algebraic simplification and solving linear inequalities as performed by mathematicians without the explicit use of the axioms of an ordered field.

A recent approach for mathematics-oriented theorem proving is *proof planning*. Proof planning was first introduced by BUNDY in 1988. BUNDY's key idea was to augment individual tactics with pre- and postconditions that specify the applicability of the tactic as well as its effects with respect to a proof state. This results in AI-planning operators, so-called *methods*. A proof planner searches for a plan, i.e., a sequence of methods, that derives the theorem from the given assumptions. The representation of a proof, at least while it is developed, consists of a sequence of abstract steps. The complete abstract proof (or parts of it) can be expanded to a logic-level proof. This enables automated proof search at an abstract level and a separated checking process.

BUNDY and his group developed the first proof planner, CLAM [44], in the early 1990s and applied it to prove theorems by mathematical induction. To guide the search of inductive proofs the rippling search heuristic for difference reduction [121, 46] is encoded into CLAM methods. Later on BUNDY and his group re-implemented CLAM in their new system λCLAM [45, 204].

Another proof planner is part of the ΩMEGA system [213]. ΩMEGA is a proof development system for knowledge-based interactive and automated proof construction developed by SIEKMANN and his group since the mid 1990s (e.g., see [118, 18]).

[2] For the diagram a Cartesian representation was used, with each point mentioned in the theorem being assigned a pair of x, y coordinates chosen in such a way as to make the assumptions of the theorem true.

The development of ΩMEGA was motivated by the conviction that the solution of main-stream mathematical problems requires the combination of theorem proving based on mathematical knowledge with powerful reasoning experts such as machine-oriented theorem provers, computer algebra systems, or constraint solvers. ΩMEGA employs proof planning as the main tool for automated proof construction since proof planning enables the incorporation of mathematical knowledge into the theorem proving process as well as the incorporation of external expert systems. Since the focus of ΩMEGA's proof planning is on the integration of mathematical knowledge it is called *knowledge-based proof planning*.

One difference between proof planning in ΩMEGA and CLAM is the handling of heuristic control knowledge. Preconditions of CLAM methods may include legal conditions about the feasibility of the application of the method as well as heuristic conditions about the desirability of the application of the method. In contrast, preconditions of ΩMEGA methods include only legal conditions. Heuristic control knowledge is encoded in so-called control rules. Technically, the control rules guide the search by reasoning on alternatives at choice points. That is, they can prefer promising alternatives and reject or delay alternatives that are not likely to lead to a solution. Thereby control rules can encode mathematical control knowledge. This is possible since, as opposed to the local and syntactic proof heuristics used in machine-oriented provers, ΩMEGA's control rules can reason about the current proof planning state as well as about the entire history of the proof planning process. Moreover, they can cover semantical information on particular mathematical functions or constants that guides human proof search. We shall give a detailed description of the ΩMEGA system in chapter 3. An introduction of knowledge-based proof planning is given in chapter 4.

A major difference between systems such as the Geometry-Theorem Proving Machine, SUMS, or IMPLY and proof planning is how knowledge is used and incorporated. Whereas the former systems are special-purpose systems in which domain-specific knowledge is hard-wired into the system, in proof planning only methods and control rules are domain-specific. The representational techniques and reasoning procedures are general-purpose.

The ΩMEGA system has been used in several case studies, which illustrate the interplay of the various components such as proof planning and external reasoning systems. The first large case study was the application of ΩMEGA's proof planning to limit problems [172]. Another class of problems we tackled with proof planning are residue class problems [165]. We also employed proof planning to solve problems of permutation groups [57] and homomorphism problems. Since they are part of this thesis we shall discuss these case studies and the knowledge we acquired and formalized to tackle them in the chapters 8 — 10. Another case study not discussed in this thesis is proof planning for diagonalization proofs [49] of theorems such as CANTOR's theorem and the undecidability of the halting problem. A case study on interactive proof development with ΩMEGA is the proof of the *Irrationality of* $\sqrt{2}$ [215, 214]. Here, the user interactively proposes the main conceptual steps. Simple but painful logical subproofs are then passed to connected machine-oriented provers and computations are done by connected computer algebra systems.

2.2 Blackboard Systems

In this section, we briefly introduce blackboard architectures. In particular, we shall describe the HEARSAY-III and the BB1 systems since they are relevant for the understanding of MULTI's blackboard architecture. An extensive introduction to blackboard systems can be found in [76].

2.2.1 Introduction to Blackboard Systems

The central issue of any kind of knowledge-based problem solving deals with the question: What piece of knowledge should be applied when and how? The “standard” computation approach is a central sequencing program that consists of a set of procedures and some control mechanisms for ordering their application. The problem-solving knowledge is embedded in the procedures and the control structure. This approach is suitable to apply procedures in a deterministic or quasi-deterministic way. However, it is not flexible enough, if many and diverse procedures have to be combined in a non-deterministic way. *Blackboard architectures* have been developed in the eighties to enable a flexible combination of different problem solving procedures in a single problem solving process and to realize a non-deterministic solution-strategy.

The fundamental ideas of the blackboard model are (1) the segmentation of the knowledge base into modules that are kept separate and independent and (2) the separation of the inference engines that work on that knowledge. Each knowledge module can employ its own inference engine. The communication between the modules is limited to reading and writing in a common working memory, the *blackboard*. The blackboard can be further structured into regions that, for instance, contain different data structures. A basic blackboard architecture consists of a structured blackboard and the modular inference engine/knowledge base pairs which are called the *knowledge sources*. Figure 2.1 depicts such a basic blackboard architecture.

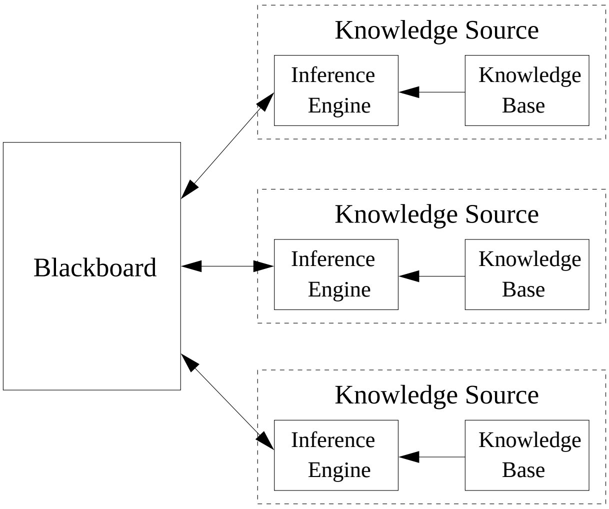


Figure 2.1: A rudimentary blackboard architecture.

The objective of each knowledge source is to contribute to the solution of the problem whose problem-solving state data are kept on the global blackboard. Control of knowledge source activation in blackboard systems is *data-directed* and *event-driven*. That is, the activation of the next knowledge source is determined by the changes of the data on the blackboard caused by other knowledge sources, rather than by explicit calls from other knowledge sources or some central sequencing mechanism. Knowledge sources check whether they are applicable with respect to the current solution state on the blackboard and indicate their applicability. Control modules choose the next knowledge source based on the solution state and on the existence of knowledge sources capable of improving the current state of the solution. As a result, the sequence of knowledge source invocation is dynamic rather than fixed and preprogrammed. The ability of a system to flexibly exploit its best data and most promising methods is also called *opportunistic problem solving* [112]. Pieces of problem solving steps occur in the following iterative sequence:

1. A knowledge source changes blackboard objects.
2. Each knowledge source indicates the contribution it can make with respect to the changed solution state.
3. Using the information produced in step 1 and 2, a control module selects the next knowledge source to become active.

With respect to step 1 and 2 knowledge sources can be seen also as *condition-action pairs*. A knowledge source contains the knowledge when it is applicable (the condition part of a knowledge source, which is employed in step 2) and how it is applicable (the action part of a knowledge source, which is employed in step 1).

The first blackboard architectures were the HEARSAY-II[77] and the HASP[181] architectures. HEARSAY-II was used for speech recognition and HASP for ocean surveillance. Both consisted of a single blackboard and a set of hierarchically structured knowledge sources. The control in HEARSAY-II is subsymbolic. Each knowledge source as well as each object on the blackboard has a *rank of belief* (a numeric value). From these values a scheduler computes and selects the most promising application of a knowledge source to an object of the blackboard. In HASP the control knowledge was organized in hierarchically structured modules that consist of sets of rules. On the lowest level is a set of knowledge sources that manipulate objects on the blackboard. At the next level there are knowledge source activators that know, when to use the various knowledge sources. On the highest level a strategy module analyzes the current solution state and selects the next knowledge source activator.

In later blackboard systems the control became are more and more important issue. Therefore, later architectures tried to make control of the system a knowledge-based procedure in its own right. In the HEARSAY-III [78] and the BB1 [111] frameworks control is established as a first-class knowledge-based activity. Both frameworks employ architectures with two separate blackboards: one blackboard to reason on the *domain problem*, that is, the given problem to solve, and one blackboard to reason on the *control problem*, that is, the problem which applicable knowledge source to apply next. Corresponding to the two separated blackboards, these systems employ also two separated sets of knowledge sources that reason about the domain problem and about the control of problem-solving actions, respectively.

Since the blackboard architecture of MULTI resembles the HEARSAY-III and BB1 architecture, we shall now introduce these two frameworks in more detail. MULTI's blackboard architecture is described in detail in section 6.2.2. A discussion of similarities and differences between MULTI and HEARSAY-III and BB1 follows in section 6.3.1.

2.2.2 The HEARSAY-III Framework

HEARSAY-III is a domain-independent architecture. The motivation for the development of HEARSAY-III was the observation that the control problem exhibits characteristics similar to the domain problem. Hence, the same blackboard-oriented knowledge-based approach should be used for its solution as well.

Hearsay-III employs two blackboards: the *domain blackboard* for the solution of the domain problem and the *scheduling blackboard* for the solution of the control problem. Each blackboard can be subdivided. Correspondingly, Hearsay-III divides the knowledge sources into *domain knowledge sources* and *scheduling knowledge sources*. All knowledge sources are condition-action pairs. The condition part states which events trigger the knowledge source. The action part describes how the content of the blackboards is changed, when the knowledge source is executed. The condition part of scheduling knowledge sources may reason about both, the content of the domain blackboard and the content of the scheduling blackboard whereas the condition part of domain knowledge sources reasons only about the domain blackboard. The action parts of scheduling knowledge sources effect only the scheduling blackboard, and the action parts of domain knowledge sources effect only the domain blackboard.

The system works as follows: when a knowledge source execution terminates, all knowledge sources check whether their condition part is satisfied by the contents of the blackboards. If this is the case, the knowledge source creates a so-called *activation record* that is stored on the scheduling blackboard. How the next activation record is chosen can be specified by the user who has to specify a so-called *base scheduler* procedure. The base scheduler is intended to be very simple since most of the knowledge about scheduling should be embodied in the scheduling knowledge sources. Moreover, the user can specify how the activation records are maintained on the scheduling blackboard by the scheduling knowledge sources. For instance, the activation records might be stored in a queue and actions of scheduling knowledge sources change this queue. The base scheduler then might consist simply of a loop that removes the first element from the queue and calls for its execution. If the queue is empty, the base scheduler terminates marking the end of system execution.

When several scheduling knowledge sources are applicable, the problem is how to schedule the scheduling knowledge sources? To deal with this problem, Hearsay-III allows for dividing the scheduling blackboard into a set of mutually exclusive, prioritized scheduling levels. Each scheduling knowledge source is assigned to a single level. The base scheduler always returns an activation record from the highest level on which activation records reside.

2.2.3 The BB1 Framework

As Hearsay-III BB1 is a domain-independent framework that can be filled by the user. Furthermore, BB1 is similar to Hearsay-III in that it distinguishes domain and control problems, blackboards, and knowledge sources. The *control problem* whose solution motivated the development of BB1 is formulated more generally than the control problem of Hearsay-III: which of its potential actions should an AI-system perform at each point in the problem solving process? Technically, the BB1 approach for control extends the Hearsay-III approach since it deals not only with the question which knowledge source to execute next but it allows also for adapting the control of the system itself, for instance, by adopting, retaining, and discarding control heuristics.

In [111] Hayes-Roth operationalizes intelligent control problem solving as the achievement of (at least) the following behavioral requirements:

- Make explicit control decisions that solve the control problem.
- Decide which actions to perform by reconciling independent decisions about what actions are desirable and which actions are feasible.

- Adopt, retain, and discard individual control heuristics in response to dynamic problem solving situations.
- Decide how to integrate multiple control heuristics of varying importance.
- Dynamically plan strategic sequences of actions.

The BB1 architecture is designed to achieve these goals. As opposed to the scheduling knowledge sources of HEARSAY-III, which reason only about the execution of other knowledge sources, the control knowledge sources of BB1 incrementally construct dynamic control plans for the systems behavior on the control blackboard. A *control plan* is a set of related control decisions that influence each other and that can be dynamically created and changed by control knowledge sources. Decisions can describe desirable actions (i.e., desirable executions of knowledge sources) and determine which of the system's control heuristics operate during particular problem solving time intervals. Different kinds of decisions are placed on different levels of the control blackboard (e.g., strategy, policy, focus decisions). In each cycle, the scheduler uses the heuristics determined by the current decisions on the control blackboard to select one of the applicable knowledge sources for execution. This can be either a domain knowledge source that works on the domain blackboard or a control knowledge source that can modify the decisions on the control blackboard.

In particular, BB1 allows to integrate the data-directed control of blackboard systems with goal-directed control (e.g., see [64, 126]). Even if the control of the scheduling in a blackboard system is very elaborate, the problem solving process is opportunistic. *Goal-directed reasoning*, in contrast, entails identifying and performing actions in order to perform and enable other actions, which may be desirable *per se* or because of their effects. Usually, blackboard systems miss goal-directed capabilities: There is no inference process to predict the effects of executing a knowledge source. Moreover, there is no process that records which preconditions of a (desirable) knowledge source are missing such that the knowledge source is not executable. Thus, it is not possible to compute sequences of related knowledge sources that achieve an important long-time goal (e.g., to solve a particular subproblem or to create the blackboard content that triggers particularly desirable knowledge source).

BB1 can initiate goal-directed reasoning in two situations: (a) the system notices that it has an important focus decision on the control blackboard, but there is no executable knowledge source that satisfies it; or (b) the system notices that it has a highly desirable knowledge source with unsatisfied preconditions. In the application scenario described in [126], a control knowledge source is triggered whenever no executable knowledge sources rate highly against an important focus decision on the control blackboard. When executed, this knowledge source determines which potential other knowledge sources could rate highly against the focus and which of their preconditions are not satisfied. Then, it posts a goal-directed focus decision for each such precondition. Another control knowledge source is triggered whenever a highly desirable knowledge source has unsatisfied preconditions. When executed, this knowledge source also posts a goal-directed focus decision for each unsatisfied precondition of this knowledge source. Then, other control knowledge sources prefer executable knowledge sources that rate highly against such a focus. Note that this reasoning process is only possible when the first two described control knowledge sources can reason on the preconditions of other knowledge sources and when the third described control knowledge source can reason on the effects of other knowledge sources. If preconditions and effects of knowledge sources can be described, then it is possible to perform planning at the level of the knowledge sources. Such an approach is described, for instance, in [75].

2.3 AI-Planning

In order to build intelligent agents that act in the world algorithms are needed for generating appropriate sequences of actions. One approach to solve this problem is *AI-planning.*

A *planning problem* consists of

1. a description of the initial state of the world in some formal language,
2. a description of the agent's goals in some formal language, and
3. a description of the possible operations that the agent can performs in some formal language.

A *planner* is an algorithm that is applied to a planning problem and returns a sequence of *actions*, i.e., instantiated operations, which will achieve the goal, when executed in any world satisfying the initial state description. Such a sequence of actions is also called a *solution plan.*

This formulation of the planning problem is very abstract. In fact, it specifies a class of planning problems parameterized by the languages used to represent the world, goals, and operations. In general, there is a spectrum of more and more expressive languages (e.g., see [241, 206]). A planning algorithm becomes more complex for more expressive representation languages, and the speed of the resulting algorithm may decrease as well.

A very simple, yet very influential language is the propositional STRIPS representation.[3] STRIPS describes the initial state of the world with a complete set of ground literals. It restricts the type of goals that may be specified to conjunctions of positive literals. Operations are represented in the STRIPS language as *operators* (also called *operator schemata*) with *preconditions* and *effects*. The preconditions of each operator have the same restriction as the problem's goals: they are a conjunction of positive literals. An operator's effects are a conjunction that may includes both, positive and negative literals. All the positive literals in the operator's effects are called the *add-list* of the operator, while all the negative literals are called the *delete-list* of the operator. A more expressive language is PDDL [155] (Planning Domain Definition Language), which is used to specify the problem sets for the planner competitions held at recent AIPS conferences [156]. PDDL allows — among others — for the specification of universal and conditional effects.

The classical approach to solve planning problems is *precondition achievement planning* [74]. Precondition achievement planning goes back to the General Problem Solver, GPS [179]. STRIPS focused and distilled the technique to the form used in planning: During the planning process, first an *unsatisfied precondition* is chosen (this condition is not true and but it should be). Then, the available operators are checked whether their add list contains an effect to achieve this precondition. One operator is chosen, appropriately instantiated (bind the variables of the operator to elements of the plan), and the resulting action is inserted into the plan under development. Then, the preconditions of the introduced action become new unsatisfied preconditions of the plan whereas the initially unsatisfied precondition is *satisfied* by an effect of the introduced action.

[3] The acronym "STRIPS" stands for "STanford Research Institute Problem Solver', a very famous and influential planner build in the 1970s to control an unstable mobile robot known as "Shakey" [86, 85].

Almost all traditional approaches in AI-planning follow the precondition achievement paradigm. *State-space planners*[4] such as Strips and Prodigy [234] as well as *plan-space planners*[5] such as Noah [207] and Ucpop [191]. Other planning approaches, e.g., *Modal Truth Criterion (MTC)* [48] and *Systematic NonLinear Planning (SNLP)* [148] differ in minor ways but also achieve a single precondition at a time and build a final solution plan by eventually achieving all operator preconditions.

The complexity of traditional precondition achievement AI-planning mainly stems from planning for *conjunctive goals*, that is, goals that consist of several facts that all have to be achieved at the same time (e.g., see [48]). Given a conjunctive goal, it seems natural to try divide and conquer, but the subplans achieving the single subgoals may interfere and do not achieve the desired goals together. A famous example for this problem is the so-called "Sussman anomaly" problem in the blocks world.[6]

This problem pushed the development of precondition achievement planners that follow a *least commitment* approach (e.g., see [241]). The idea of least commitment approaches is to delay decisions as long as possible. For instance, decisions on the order of actions can often be delayed until finally a solution plan, i.e., a sequence of actions, has to be computed. Noah was the first system that introduced *partial-order planning* in which plans can be assembled as partial orders rather than total orders of actions. Often set of constraints (e.g., ordering constraints) are used to represent sets of possible solutions plans. The constraint that a precondition p of a certain action A is achieved by an effect of another action A' and should be preserved between the execution of A and A' is expressed by so-called *causal links* [191, 241] or *interval preservation constraints* [129, 128]. The validity of such constraints is potentially threaten by an action A'' that has a negative effect p. A'' cannot be executed between A or A' since it would remove the effect p of A that is needed for A'. A solution is to execute A'' before A or after A'. These techniques to resolve *threats* are called *promotion* and *demotion*, respectively.

In the last years, several new planning techniques have been developed:

Graphplan The two-phase *Graphplan algorithm* [32] first stores all possible actions and potentially satisfied preconditions up to a certain depth in a planning graph. Afterwards, the Graphplan algorithm alternates between two phases: solution extraction and graph expansion. The solution extraction phase searches in the current planning graph for a plan. If no solution is found, then the graph expansion phase extends the planning graph by adding further levels of actions and potentially satisfied preconditions. Systems that use a Graphplan algorithm are GraphPlan, IPP [139], and STAN [88].

SAT Methods Another more recent approach [132] compiles planning problems into a propositional formula, which, if satisfiable, implies the existence of a solution plan. In order to obtain a satisfying assignment, systems such as SATPLAN [132] use speedy systematic or stochastic satisfiability methods.

Combination of Graphplan with other methods The Graphplan representations form the basis of several encodings of planning problems into other

[4] State-space planners search the space of possible world states. That is, each node in the search space denotes a state of the world, and links connect world states that can be reached by executing a single action.

[5] Plan-space planners search the space of possible (partial) plans. That is, each node in the search space denotes a partial plan, and links connect partial plans that can be reached by introducing a single action.

[6] A detailed discussion of planning in the blocks world can be found in standard AI-textbooks, e.g., in [206].

formalizations. These approaches replace the solution extraction phase of the Graphplan algorithm by a transformation into a different formalism and the application of algorithms specialized for this formalism. For instance, the BlackBox [133] system combines Graphplan and SAT methods. It encodes the planning graph into a propositional formula to which it applies SAT methods. Another example is the GP-CSP system [72], which combines Graphplan and constraints satisfaction problems (CSP). Here, the planning graph is converted into a CSP encoding to which standard CSP solvers are applied.

Heuristic Planning A different approach interprets planning as heuristic search [154, 24]. Heuristic planning is based on the ideas of heuristic search [182, 190] and is similar to the search in problems as the 8-Puzzle. The difference is in the heuristic: while in problems as the 8-Puzzle the heuristic is typically given (e.g., as the sum of Manhattan distances), in planning it is extracted automatically from the declarative representation of the problem. Heuristic planners perform a state-space *regression* or *progression search*[7] and use well-known search algorithms that are guided by the heuristic. For instance, the Hsp system [24] searches the progression space with a hill-climbing algorithm. FF [116] searches also the progression space using a different hill-climbing algorithm. HspR* [110] searches the regression space using the IDA* algorithm.

These approaches yield extremely speedy planners, which are in many cases orders of magnitude faster than systems following the precondition achievement approach. However, it is an open question how well these approaches are able to deal with complex real world problems. Indeed, the application successes of planning systems such as Sipe [243] and O-Plan [186] are due to — among others — hierarchical abstraction in planning and domain knowledge. First, a plan is constructed at an abstract level. Then, this abstract plan is successively refined by expanding actions and re-planning. An expansion can replace a single action with an entire plan fragment. Technically, *hierarchical task network (HTN) planning* [229] distinguishes *primitive actions* and *non-primitive actions* (e.g., see [79]). Non-primitive actions are replaced by reduction schemas, i.e., plan fragments consisting of other abstract or primitive actions, until a sequence of primitive actions is constructed. Action sequences containing primitive actions only are executable. Drummond [74] and Wilkins [244] argue that the superiority of these systems in real world applications[8] stems from the possibility to encode more domain knowledge into the planning process, in particular, to formulate the domain knowledge more naturally in terms of pre-packaged plan fragments.

[7] State-space progression planning searches forwardly in the space of states. It starts with the initial state. Given a current state, the next state in the search space is computed by simulating the execution of an action whose preconditions are satisfied in the current world state. The process stops as soon as a state is reached, which satisfies all goals. State-space regression planning searches backwardly in the space of states. It starts with a *goal-conjunction* consisting of all given goals. Such a goal-conjunction represents the set of all states that satisfy at least all the elements of the conjunction. Given a current goal-conjunction, the next goal-conjunction (representing the next set of states) results from the introduction of an action by adding all preconditions of the action and removing all effects of the action. The process stops if the initial state satisfies all elements if the goal-conjunction, that is, if the initial state is in the set of states represented by the goal-conjunction. For further details on state-space progression and regression planning see [241, 182, 237].

[8] Examples for real-world applications of these systems are: the application of Sipe for controlling beer production [242], and the application of O-Plan to the problem of spacecraft mission planning [65].

Chapter 3

An Introduction to ΩMEGA

The ΩMEGA proof development system [213] is at the core of several related and integrated research projects of the ΩMEGA research group, whose aim is to develop system support for the working mathematician. By providing tactics for interactive proof development ΩMEGA has many characteristics in common with systems such as NUPRL [3], ISABELLE [189], HOL [107], and PVS [188]. However, it differs significantly from these systems with respect to its focus on *proof planning* (introduced in chapter 4) for automated and mathematics-oriented proof development and in that respect it is more similar to the systems CLAM and λCLAM developed at Edinburgh [45, 204]

The ΩMEGA system combines interactive and automated proof construction for domains with rich and well-structured mathematical knowledge. The inference mechanism at the lowest level of abstraction is an interactive theorem prover based on a higher-order natural deduction (ND) variant of a soft-sorted version of Church's simply typed λ-calculus [54]. While this represents the "machine code" of the system the user will seldom have to see, the search for a proof is usually conducted at a higher level of abstraction defined by *tactics* and *methods*. Proof construction is also supported by already proved assertions and lemmas and by calls to external systems to simplify or solve subproblems.

At the core of ΩMEGA is the *proof plan data structure* ($\mathcal{PDS}$) [50] in which proofs and proof plans are represented at various levels of granularity and abstraction. The proofs and proof plans are developed with respect to a taxonomy of mathematical theories, which is currently being replaced by the mathematical data base MBASE [89, 141]. The user of ΩMEGA, the proof planners MULTI and PLAN, or the suggestion mechanism Ω-ANTS modify the $\mathcal{PDS}$ during proof development. They can also invoke external reasoning systems whose results are included into the $\mathcal{PDS}$ after appropriate transformation. Once a complete proof at the most appropriate level of abstraction has been found, this proof can be expanded to lower levels of abstraction until finally, a proof at the level of the logical calculus is established. After expansion of these high level proofs to the underlying ND-calculus, the $\mathcal{PDS}$ can be checked by ΩMEGA's proof checker.

Hence, there are two main tasks supported by this system:

1. to find a proof at an abstract level,

2. to expand this proof into a calculus-level proof.

And both jobs can be equally difficult and time consuming.

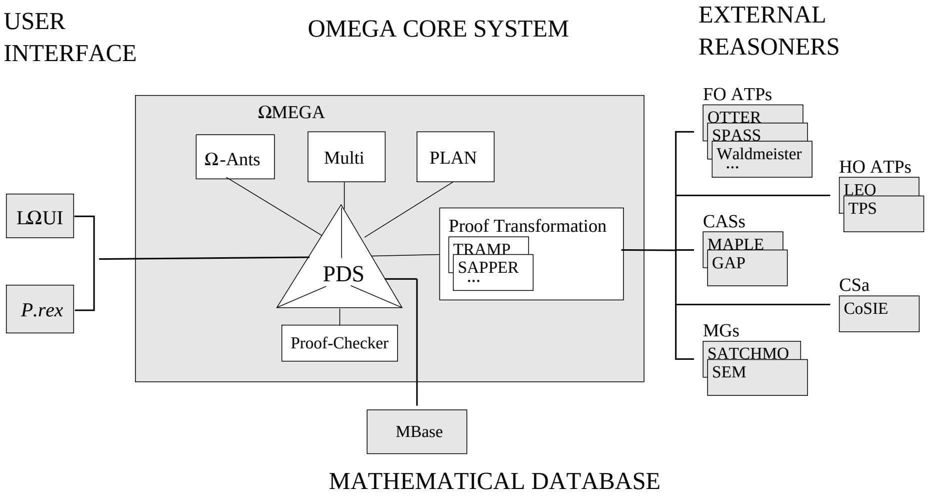


Figure 3.1: The architecture of the ΩMEGA proof assistant. Thin lines denote internal interfaces and thick lines denote internet communication via MATHWEB-SB.

User interaction is supported by the graphical user interface $\mathcal{L}\Omega\mathcal{UI}$ [109] and the interactive proof explanation system P.REX [84].

Figure 3.1 illustrates the basic architecture of ΩMEGA. ΩMEGA consists of several independent modules. These modules are connected via the mathematical software bus MATHWEB-SB [256]. An important benefit is that MATHWEB-SB modules can be distributed over the Internet and are accessible by other distant research systems as well.

This thesis describes proof planning with multiple strategies, which is realized in the MULTI system. MULTI is implemented as a component of the ΩMEGA core system as depicted in Figure 3.1. Currently, a user of ΩMEGA can apply both systems, MULTI and PLAN, the previous proof planner of ΩMEGA. However, since MULTI is a considerable progress over PLAN and PLAN is not longer maintained, MULTI will be the only proof planning device in new distributions of ΩMEGA.

In this chapter, we describe the parts of ΩMEGA relevant for this thesis. We start with a section that briefly introduces ΩMEGA's logic, i.e., its syntax, its semantics, and its natural deduction calculus.[1] Then, we explain proof construction in ΩMEGA, including ΩMEGA's tactical theorem proving and a brief description of the $\mathcal{PDS}$ and the ΩANTS mechanism. The next chapter contains a detailed description of ΩMEGA's knowledge-based proof planning including an introduction of PLAN.

3.1 ΩMEGA's Logic

ΩMEGA's basic logic is a higher-order logic based on a simply typed lambda calculus. Proofs are constructed in a natural deduction calculus of GENTZEN [96] and PRAWITZ [198] . In the following, we first introduce the syntax and semantics of the logic and then we give the inference rules of the natural deduction calculus. Soundness and (Henkin) completeness of a variant of ΩMEGA's higher-order natural deduction calculus are addressed in [17].

[1] ΩMEGA's logic was first formally described in the PhD thesis of Volker Sorge [223]. The content of this section is a slightly revised version of section 2.1 in [223].

3.1.1 Syntax

Definition 3.1 (Types): Let $\mathcal{T}_B$ be a nonempty, finite set of symbols. The set $\mathcal{T}$ of *types* is defined inductively as the smallest set containing $\mathcal{T}_B$ and all types of the form $\alpha \to \beta$ where $\alpha, \beta \in \mathcal{T}$.
We call the elements of $\mathcal{T}_B$ *base-types* and types of the form $\alpha \to \beta$ *functional types*. □

In the sequel, we assume a fixed set of base-types $\mathcal{T}_B$ and types $\mathcal{T}$ with $\{o, \iota\} \subset \mathcal{T}_B$ where 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 use small Greek letters for the syntactical variables denoting elements of $\mathcal{T}$.

Notation 3.2: $\to$ associates to the right. Thus, $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \beta$ 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 introduced.

Definition 3.3 (Typed sets): A family of sets of symbols $\Gamma = (\Gamma_\alpha)_{\alpha\in\mathcal{T}}$ is called a *typed collection of sets* over $\mathcal{T}$. We call Γ *disjoint* if $\Gamma_\alpha \cap \Gamma_\beta = \emptyset$ holds for $\alpha \neq \beta$ and $\alpha, \beta \in \mathcal{T}$.
The mapping $\tau\colon \Gamma \to \mathcal{T}$ 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 $\mathcal{M}_\mathcal{T} = (\mathcal{M}_\alpha)_{\alpha\in\mathcal{T}}$ for $\mathcal{M}_\alpha = \{f|\tau(f) = \alpha\}$.
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 or has been explicitly stated earlier, for instance, in definitions of defined symbols.

Definition 3.4 (Signature): Let Σ be a disjoint typed collection of 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 universal quantifier, respectively. They are just like the first-order standard versions but appear in the simply typed higher-order fashion. $\imath$ is Bertrand Russell's iota-operator in higher-order fashion as used 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 3.1.3.

Note that the universal quantifier $\Pi_{\alpha oo}$ and the description operator $\imath_{\alpha o\alpha}$ in definition 3.4 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*.

With the preceding definitions we can 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, with polymorphic definitions in most cases we can state the elements of Σ in a finite way even it is a collection of infinite sets.

Definition 3.5 (Well-formed formulas): Let Σ be a signature over $\mathcal{T}$ and $\mathcal{V}$ a collection of typed sets over $\mathcal{T}$ with infinitely many elements. We call $\mathcal{V}$ the *set of* *typed variables*. For each type $\alpha \in \mathcal{T}$ we inductively define the family $(\mathbf{wff}_\alpha(\Sigma))_{\alpha\in\mathcal{T}}$ of *well-formed formulas* by

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

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

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

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

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 $\lambda X. \mathbf{A}$ λ-*abstractions* or simply *abstractions*. The elements of $\mathbf{wff}_o(\Sigma)$ will be called *propositions*.

Notation 3.6: In the tradition of [5] the square dot '.' in $\lambda X. \mathbf{A}$ separates 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 3.7: 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 3.8 (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 $\mathbf{A}$ if and only if it is in a subformula of the form $\lambda Z. \mathbf{B}$ 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 3.9 (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) $\lambda X. \mathbf{A} \to_\alpha \lambda Y. [Y/X]\mathbf{A}$, provided Y does not occur in $\mathbf{A}$ (α-*conversion*)

(ii) $(\lambda X. \mathbf{A})\mathbf{B} \to_\beta [\mathbf{B}/X]\mathbf{A}$, provided no λZ occurs in $\mathbf{A}$ such that Z occurs in $\mathbf{B}$ (β-*reduction*)

(iii) $(\lambda X. \mathbf{A}X) \to_\eta \mathbf{A}$, 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 subterms in given terms.

Definition 3.10 (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 set $pos(t)$ of *term positions* in t is inductively defined as follows:

- If $t = c$ then $pos(t) = \{\epsilon\}$,
- if $t = (t_0\, t_1 \dots t_n)$ then $pos(t) = \{\epsilon\} \cup \bigcup_{i=0}^{n}\{i.p|p \in pos(t_i)\}$,
- if $t = \lambda x\mathbf{.}t'$ then $pos(t) = \{\epsilon\} \cup \{0.p|p \in pos(t)\}$,

where '.' denotes the concatenation of words in $\mathbb{N}^*$.

The *subterm* s of t at position $p_t(s) \in pos(t)$ is denoted as $s = t/p_t(s)$ and is inductively defined as follows:

- if $p_t(s) = \epsilon$ then $s = t$,
- if $p_t(s) = i.p$ and $t = (t_0\, t_1 \dots t_n)$ then $s = t_i/p$,
- if $p_t(s) = 0.p$ and $t = \lambda x\mathbf{.}t'$ then $s = t'/p$.

We write term positions in brackets as $\langle \pi.\tau \rangle$, where $\pi, \tau \in \mathbb{N}^*$. □

3.1.2 Semantics

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

Definition 3.11 (Frame): A *frame* $\mathcal{D}$ is a collection of nonempty sets $\mathcal{D}_\alpha$, one for each type symbol α such that $\mathcal{D}_o = \{\top, \bot\}$ and $\mathcal{D}_{\alpha\to\beta} \subseteq \mathcal{F}(\mathcal{D}_\alpha \to \mathcal{D}_\beta)$, where $\mathcal{F}(\mathcal{D}_\alpha \to \mathcal{D}_\beta)$ 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 3.12 (Interpretation of constants): Given a frame $\mathcal{D}$ 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 3.4.

Definition 3.13 (Interpretation of logical constants): Given the logical constants $\{\neg, \vee, \Pi^\alpha, \imath^\alpha\} \subseteq \Sigma$ from definition 3.4, we restrict the interpretation $\mathcal{I}$ in the following way:

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

(ii) $\mathcal{I}(\vee)(d, e) = \top$ if and only if $d = \top$ or $e = \top$, $d, e \in \mathcal{D}_o$

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

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

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 3.7.

Although the logical constants from definition 3.13 are sufficient to define a proper logic, for notational convenience we enrich our signature by addition of the following abbreviations[2]:

[2] 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.

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

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 3.4 the definition of $\doteq^\alpha$ is polymorphic.

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

Definition 3.14 (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 3.15 (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) $\mathcal{I}_\varphi(\lambda X_\alpha\text{.}\mathbf{A}_\beta)$ as the function in $\mathcal{D}_{\alpha\beta}$ such that for all $z \in \mathcal{D}_\alpha$ holds: $(\mathcal{I}_\varphi(\lambda X_\alpha\text{.}\mathbf{A}))z := \mathcal{I}_{\varphi,[z/X]}(\mathbf{A})$. □

Given our definition of a frame so far, we cannot be sure that the function required in definition 3.15 (iv) exists in $\mathcal{D}_{\alpha\beta}$. The domain $\mathcal{D}_{\alpha\beta}$ might be too sparse [4]. Because of the inductive nature of the definition this problem also affects 3.15 (iii). However, in the semantical domains of interest — the Henkin models [113] — this possibility is explicitly excluded; that is, every formula in $\mathbf{wff}(\Sigma)$ can be denoted.

Definition 3.16 (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 evaluate propositions.

Definition 3.17: 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) **P** is called *Henkin-valid* or a *Henkin-tautology* if **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 **M**, provided there is some assignment φ such that $\mathcal{I}_\varphi(\mathbf{P}) = \top$ for all $\mathbf{P} \in \Gamma$.

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

Notation 3.18: To simplify the notation given in definition 3.17 we shall write $\Gamma \models \mathbf{P}$ to indicate that **P** Henkin-follows semantically from the set of propositions Γ and $\models \mathbf{P}$ if **P** is a Henkin-tautology.

The Henkin models given in definition 3.16 are also called *generalized models* since they still allow for incomplete domains (even with the restriction we discussed with respect to definition 3.15):

$$\mathcal{D}_{\alpha \to \beta} \subseteq \mathcal{F}(\mathcal{D}_\alpha \to \mathcal{D}_\beta). \qquad (3.1)$$

This means that the set of all Henkin-valid formulas is only a subset of the set of all (standard-) valid formulas. Based on the notion of Henkin models we can define the *standard models* by requiring

$$\mathcal{D}_{\alpha \to \beta} = \mathcal{F}(\mathcal{D}_\alpha \to \mathcal{D}_\beta). \qquad (3.2)$$

Thus, the standard models form a subclass of the Henkin models, and the set of valid formulas in an arbitrary Henkin model is generally smaller than the set of valid formulas in the standard models. However, GÖDEL showed in his *incompleteness theorem* that there exists no calculus that is both sound and complete for standard validity, whereas it was proved by HENKIN in 1950 that complete and sound calculi can be constructed for Henkin validity.

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, 15] for a more detailed introduction and examination of this subject.

3.1.3 Calculus

The original *natural deduction* (ND) calculus was introduced by GENTZEN [96] in 1935. The idea is to model mathematical problem solving behavior in small logical steps for a 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 higher-order variant of GENTZEN's classical ND-calculus.

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 3.13 and to have the subsequently defined abbreviations available. Although confining ourselves to the original logical constants from definition 3.4 would result in a leaner calculus, we prefer a more expressive and intuitive basic calculus by also allowing for inference rules for the abbreviations available. However, the larger the basic calculus is, the less efficient it is to check proofs automatically. Therefore, we will not allow for equality and equivalence as primitive concepts and rather define them as derived concepts (see section 3.2.1).

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

Definition 3.19 (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) $\begin{array}{c}[A]\\ \vdots\\ B\end{array}$ 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 quodlibet*). While all these rules are basically first order we have also one proper higher order rule that performs λ conversions.

Definition 3.20 (Inference rules): Given propositions $P, Q, R \in \mathbf{wff}_o(\Sigma)$ we can define the inference rules of the natural deduction calculus as given in Figure 3.2.

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. $[c/x]$ means that the term has to be a constant. 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 constant c does not already occur in the proposition P in case of the $\forall_I$ rule. In the $\exists_E$ rule the constant c 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 proof assumption that is equal with respect of the λ-conversions given in definition 3.9; that is, A denotes the same term as B up to $\beta\eta$-reduction and renaming. Additionally, we introduce the rule *Weaken*, which is a special case of the $\lambda\!\leftrightarrow$ rule since it allows to justify a goal with an assumption containing the same formula meaning they are trivially equal. Although *Weaken* does not increase the expressivity of the basic calculus, it is a useful rule for proof construction.

$$\frac{A}{A}\ Weaken$$

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

Definition 3.21 (Axioms): We define the following four axioms for our calculus:

- $\forall A_o \centerdot A \vee \neg A$ (*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)

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

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

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

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

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

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

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

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

Figure 3.2: The inference rules of the natural deduction calculus.

- $\forall A_o \boldsymbol{.} \forall B_o \boldsymbol{.} (A \Leftrightarrow B) \Rightarrow (A \doteq B)$ (Boolean extensionality)
- $\forall P_{\alpha o} \boldsymbol{.} \exists X_\alpha \boldsymbol{.} [PX \wedge [\forall Y_\alpha \boldsymbol{.} PY \Rightarrow [X = Y]]] \Rightarrow P(\text{℩}P)$ (Description)

□

The axiom of description in the preceding definition gives us a more precise understanding of the description operator as a function with a fixed interpretation on singleton sets (on other sets also other interpretations are possible). It expresses that for every set $P_{\alpha o}$ that contains exactly one 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, even for the Leibniz equality (which is in general weaker than primitive equality in the model and which defines equality in ΩMEGA) the respective reverse directions can be inferred 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 3.2.1).

Definition 3.22 (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 [227, 228] that *cut-elimination* holds for higher order calculi with extensionality, it is still an open problem whether appropriate cut-elimination algorithms terminate. (See also [192] for a discussion on cut-elimination in type theory.) A possible *cut rule* for our natural deduction calculus is of the form

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

which is essentially *modus barbara*. Indeed ΩMEGA offers a way to introduce cuts by having modus barbara as a tactic available (see section 3.2.2 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 3.23 (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 presentation.

Example 3.24:

The linearized natural deduction proof for the assertion:

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

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	$\vdash [P(A) \Rightarrow Q(A)]$	$(\forall E$ L1)
L5.	L1,L3	$\vdash Q(A)$	$(\Rightarrow E$ 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.		$\vdash \forall X_\iota . (P_{\iota o}(X) \Rightarrow Q_{\iota o}(X)) \Rightarrow (\forall X_\iota . P(X) \Rightarrow \forall X_\iota . Q(X))$	$(\Rightarrow I$ L2)

The same proof in tree representation:

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

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

3.1.4 Soft Sorts

The syntax of ΩMEGA's logic is extended by a sort concept. This, however, is not a full-grown sort concept as given in the literature (for instance, by SCHMIDT-SCHAUSS in [208] in the context of first order logic and by KOHLHASE for higher order logic in [140]). Instead it is a conservative extension of the logic given in the preceding section by simply introducing sorted quantifications.

Instead of having a full-fledged sort system, ΩMEGA only permits the use of so-called *soft sorts*; that is, quantified variables are relativized to a set, which is the range of the possible instantiations of the variable. This set is 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:

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

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

Using soft 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, soft sorts add to the readability of the logic since they allow to state formulas of theorems and problems more concisely. As an example consider the following statement for integers

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

which is relatively concise using sorted quantifiers. It becomes much less readable if we do not use abbreviations:

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

3.2 Proof Construction in ΩMEGA

For a given *theorem* and its *assumptions* a proof can be constructed by successively applying the ND-rules introduced in the previous section. The rules can be applied either backward or forward. In the former case, ND-rules are applied to the theorem, resulting in the introduction of the premises of the rule as new open nodes. If an applied rule has more than one premises, the problem is split into several subproblems, which have to be shown. In the latter case, rules are applied to the proof assumptions, and the conclusions of the rule are introduced as new nodes into the proof. For many applications it is interesting to mix forward and backward reasoning.

Although ΩMEGA relies on the natural deduction calculus introduced in the preceding section and although it enables proof construction with ND-rules, it's main goal is to support proof development at a more user-friendly level of abstraction. Therefore, ΩMEGA employs tactics for interactive proof development and methods for automated proof planning. Moreover, proofs in ΩMEGA are always constructed with respect to a taxonomy of *mathematical theories*. These theories provide defined concepts, their axiomatization, and already proved theorems, that can be incorporated into proofs.

To enable the use of abstract tactics and methods and their combination with calculus rules, proofs in ΩMEGA are actually constructed in a *generalized natural deduction proof* where justifications can be ND-rules (see preceding section) and also tactics, methods, as well as applications of external systems. However, for a proof to be valid in ΩMEGA it needs to be refined to a calculus-level natural deduction proof. Therefore, abstract justifications have to be expandable to calculus-level subproofs. This expansion can be recursive, 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 data structure, the so-called *proof plan data structure* $\mathcal{PDS}$.

In the sequel, we first describe how facts from the knowledge base can be incorporated into a proof object. Then, we introduce ΩMEGA's tactical theorem proving. Finally, we give brief descriptions of the proof plan data structure $\mathcal{PDS}$ and the suggestion mechanism ΩANTS.

3.2.1 Employing Facts from the Knowledge Base

Proofs in ΩMEGA are always constructed within the context of a mathematical theory. ΩMEGA's theories are hierarchically structured and connected by a simple inheritance mechanism. A theory contains defined concepts as well as axioms and theorems.

Definitions Definitions in ΩMEGA are used as definitions in a mathematical textbook: The introduction of abbreviations for complex concepts allows to shorten formulas and proofs. However, if necessary the abbreviation can be expanded by its actual meaning.

A definition is a pair consisting of the symbol that is defined (also called the *definiendum* of the definition) and a λ-term that describes the complex concept that is abbreviated (also called the *definiens* of the definition). We write a definition $(definiendum, definiens)$ as $definiendum \equiv definiens$ where $\equiv$ is called the *definition symbol*.

For instance, *equality* and *equivalence* are defined concepts in ΩMEGA's theories.

Their respective definitions in the knowledge base are of the form

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

Other defined concepts in ΩMEGA's knowledge base are, for instance, basic notions of set theory, such as the element property, the union of two sets, or the subset property, which are defined as

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

If a theorem is proved with respect to a certain theory then the defined concepts of this theory and inherited concepts can be used to formalize the problem. For instance, in a theory that comprises the concepts $\doteq$, $\Leftrightarrow$, and $\subseteq$, we can state the theorem that two sets are equal iff they are subsets of each other by the formula:

$$\forall X_{\alpha o} . \forall Y_{\alpha o} . (X \doteq Y \Leftrightarrow (X \subseteq Y \wedge Y \subseteq X)) \qquad \text{(I)}$$

During a proof attempt it is sometimes necessary to expand defined concepts by their actual definition or to contract occurrences of definitions to occurrences of the corresponding defined concepts. To establish this interface to the theory knowledge base ΩMEGA employs two extra calculus rules:

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

$\equiv_E$ 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 subterm 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 *definiendum* as a parameter in the justification.

To illustrate the concept of definition expansion consider the theorem in (I). The application of the rule $\equiv_I$ with respect to the first occurrence of the defined concept $\subseteq$ results in the formula

$$\forall X_{\alpha o} . \forall Y_{\alpha o} . (X \doteq Y \Leftrightarrow ([\lambda U_{\alpha o} . \lambda V_{\alpha o} . \forall x_\alpha . U(x) \Rightarrow V(x)](XY) \wedge Y \subseteq X))$$

Applying β-reduction to this term yields

$$\forall X_{\alpha o} . \forall Y_{\alpha o} . (X \doteq Y \Leftrightarrow ((\forall x_\alpha . X(x) \Rightarrow Y(x)) \wedge Y \subseteq X))$$

Axioms and Theorems *Axioms* in theories are facts that are stated without a proof. They allow to "axiomatize" theories or concepts. As opposed thereto, *theorems* are facts for which a valid proof has already been derived in ΩMEGA. They enable the reuse of already proved results during the proof construction for new problems. Technically, both axioms and theorems are pairs consisting of a name and a formula.

Axioms and theorems can be directly imported into a proof as so-called *theory assertions* or simply *assertions* and can be used like any assumptions of the proof. To establish this interface to the theory knowledge base ΩMEGA employs the extra calculus rule *Assertion*

$$\frac{}{Ass}\ Assertion(Ass)$$

which introduces an assertion *Ass* into the proof object under construction.

The following proof involves the application of the *Tertium non datur* (TND) axiom. The proposition to prove is $(P \Rightarrow Q) \Rightarrow (\neg P \Rightarrow Q)$ given in line *Thm*. The axiom is imported into the proof in line L_1.

L_1.		$\vdash \forall A_o \mathbin{\bullet} A \vee \neg A$	$(Assertion\ (TND))$
L_2.		$\vdash P \vee \neg P$	$(\forall_E\ L_1\ P)$
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	$\vdash \neg P \vee Q$	$(\vee_E\ L_2\ L_6\ L_8)$
Thm.		$\vdash (P \Rightarrow Q) \Rightarrow (\neg P \Rightarrow Q)$	$(\Rightarrow_I\ L_9)$

3.2.2 Employing Tactics for Proof Construction

So far, we applied calculus rules to construct proofs (see example 3.24). However, the style of calculus-level proofs produced in the previous sections is unnatural and too "low level" for many applications. Thus, many interactive systems use tactical theorem proving for complex and more abstract proofs (c.f., NUPRL [62], ISABELLE [189], HOL [107], COQ [63], QUODLIBET [144]). The idea in tactical theorem proving is that repeatedly occurring sequences of inference steps are encapsulated into macro steps, so-called tactics. The tactics enable interactive proof construction at a higher level of abstraction.

The notion of a *tactic* was invented by MILNER in the early 1970s for goal oriented, that is, in natural deduction backward theorem proving (e.g., see [175]). Essentially, a tactic is a function that does two things:

1. Splits a goal into subgoals.
2. Keeps track of the reasons why solving the subgoals will solve the original goal.

Most tactic-based theorem proving systems (e.g., NUPRL, ISABELLE, HOL) are descendants of LCF [106] and follow a bottom-up approach for tactic construction. That is, more and more complex tactics are built by combining sequences of calculus rules or other tactics with so-called *tacticals* such as THEN, ORELSE, REPEAT. For instance, the tactic REPEAT(*tac*) applies the tactic *tac* repeatedly to a goal and its subgoals. The application of such a tactic constructed in a bottom-up manner results in a sequence of calculus rules; that is, the tactic immediately expands (via several levels of tactics) to the calculus rule level during its application. In this case, the application of a tactic (if it succeeds) is *a priori* correct, given the correctness of the underlying base calculus.

In ΩMEGA, we follow a top-down approach for constructing tactics. A *tactic* is a pair of two procedures: the *derivation procedure* that performs derivations in a proof and the *expansion procedure* that expands applications of the tactic. In the remainder of this thesis, we shall use the expression *application of a tactic* to refer to the application of the derivation procedure to a certain proof situation and the expression *expansion of a tactic application* to refer to the application of the expansion procedure to a step in a proof justified by an application of the tactic. Applications of tactics can be seen as a generalized form of calculus rules application and we state them in the same format in proof trees. A difference between tactics and the calculus rules is that tactics can have multiple conclusions.

Similar to ND-rules tactics can be applied backward and forward. In the former case, the derivation procedure is applied to an open line and computes the premises of the tactic application, which are introduced as new open lines. The initial open line, which is the conclusion of the step, is closed by the application of the tactic to the premises. In the latter case, the derivation procedure is applied to some premises and computes the conclusions of the step, which are introduced as new closed lines. The new lines are justified by the application of the tactic to the premises. It is possible to specify even more application directions for a tactic (see section 3.2.4). Technically, the derivation procedure consists of subprocedures for the desired application directions. The application direction of a tactic does not matter anymore in the finished proof and for the expansion, that is, there is only one expansion procedure.

ΩMEGA's top-down definition of tactics enables the specification of quite powerful and abstract proof steps. However, in contrast to LCF-style tactics, ΩMEGA's tactics are not necessarily always correct, since the high level of abstraction in mathematically motivated tactics of sufficient generality does not allow for the specifications of all details that are ultimately required for the use of such tactics in a concrete case. For instance, ΩMEGA's tactics can employ computer algebra systems to perform computations. However, *a priori* there is no guarantee that these computations are correct since the application of a tactic in ΩMEGA is not immediately decomposed into a sequence of single calculus rule steps. Hence, the correctness of a tactic application has to be ensured *a posteriori*. This is done by expanding tactic applications. The application of the expansion procedure to a proof step that is justified by a tactic application results in a more fine-grained subproof of the tactic's conclusions from its premises. The expansion can be recursive in the sense that the introduced proof attempt can again employ abstract tactics, which have to be expanded in turn. The expansion is successful, when this process terminates with a proof at the calculus-level, which can be machine-checked. However, it is possible to employ uncertain steps within tactics (e.g., computations by a computer algebra system) whose expansion might fail.

Example 3.25: A rather simple example of a tactic in ΩMEGA and its expansion is the $\forall_I *$ tactic. The purpose of this tactic is similar to that of the $\forall_I$ rule but where $\forall_I$ removes exactly one universal quantifier $\forall_I *$ removes arbitrary many universal quantifiers.

When $\forall_I *$ is applied backward to the open line L_1

$L_1.$ $\mathcal{H}$ $\vdash \forall x_\iota . \forall y_\iota . \forall z_\iota . P_{o\iota\iota\iota}(x, y, z)$ *(Open)*

with the three terms t_1, t_2, t_3 as parameters then its derivation procedure computes the formula $P(t_1, t_2, t_3)$ in which the universally quantified variables are replaced by the terms t_1, t_2, t_3. Moreover, it introduces this formula as new open line L_2 and justifies L_1 by the application of $\forall_I *$ to L_2.

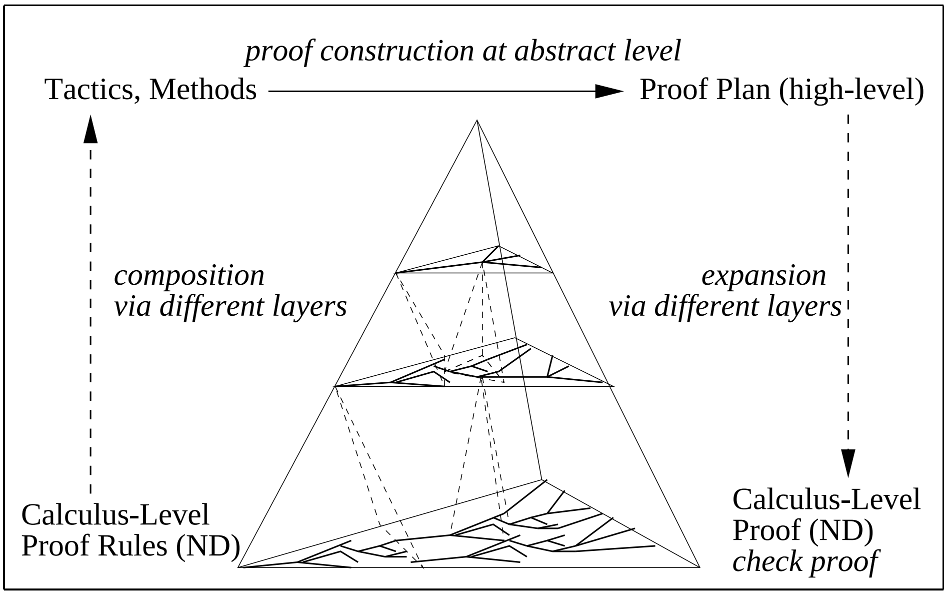


Figure 3.3: The Proof plan data structure ($\mathcal{PDS}$).

L_2.	$\mathcal{H}$	$\vdash P(t_1,t_2,t_3)$	$(Open)$
L_1.	$\mathcal{H}$	$\vdash \forall x. \forall y. \forall z. P(x,y,z)$	$(\forall_I{*}\ L_2\ (t_1,t_2,t_3))$

When this application of $\forall_I*$ is expanded, then the expansion procedure of $\forall_I*$ computes a proof segment that derives L_1, the conclusion of the application of $\forall_I*$, from L_2, the premise of the application of $\forall_I*$, with a sequence of applications of the ND-rule $\forall I$.

L_2.	$\mathcal{H}$	$\vdash P(t_1,t_2,t_3)$	$(Open)$
L_3.	$\mathcal{H}$	$\vdash \forall z. P(t_1,t_2,z)$	$(\forall I\ L_2\ (t_3))$
L_4.	$\mathcal{H}$	$\vdash \forall y. \forall z. P(t_1,y,z)$	$(\forall I\ L_3\ (t_2))$
L_1.	$\mathcal{H}$	$\vdash \forall x. \forall y. \forall z. P(x,y,z)$	$(\forall I\ L_4\ (t_1))$

3.2.3 The Proof Plan Data Structure ($\mathcal{PDS}$)

The central data structure for the overall proof construction in ΩMEGA is the *proof plan data structure* $\mathcal{PDS}$ [50] . All components of the ΩMEGA system that construct proofs work on the $\mathcal{PDS}$, for instance, the ΩANTS suggestion mechanism (see section 3.2.4) and the proof planners PLAN and MULTI.

The $\mathcal{PDS}$ is a hierarchical data structure that represents a (partial) proof attempt at different levels of abstraction. This is necessary since the inferences used for proof construction in ΩMEGA can be at different levels of abstraction. In particular, for a proof attempt to be valid in ΩMEGA it needs to be expanded into a calculus-level natural deduction proof. Hence, as opposed to other proof objects that are just planar graphs, the $\mathcal{PDS}$ has a three-dimensional structure that allows to represent direct correspondences between abstract proof steps and concrete calculus-level proofs.

Figure 3.3 depicts schematically the composition of the $\mathcal{PDS}$. Technically, the $\mathcal{PDS}$ is an acyclic graph whose nodes are proof nodes and whose edges link proof nodes that are connected by justifications using ND-rule, tactic, or method applications. One proof node can have different justifications at different levels of abstraction. Conceptually, each *abstract justification* (i.e., a justification that uses a tactic or a method) represents a subproof (the expansion of the justification) at a lower level of abstraction that is computed, when the tactic is executed.

For instance, after the expansion, the node L_1 in example 3.25 has two justifications. At the upper layer it has the justification ($\forall_I *\ L_2\ (t_1, t_2, t_3)$); the expansion of this upper layer justification results in a lower layer proof for L_1 in which it has the justification ($\forall I\ L_4\ (t_1)$). Note that the formulas of the nodes stay the same on all levels of abstraction. Thus, the $\mathcal{PDS}$ allows for derivational abstraction but not for abstraction of the objects of the logic.

3.2.4 The Suggestion Mechanism ΩANTS

The ΩANTS system was originally conceived to support interactive theorem proving in ΩMEGA [21, 22]. It provides the user with suggestions about which inference steps are applicable in the actual proof situation such that the user does not have to search painstakingly for applicable steps. Recent research aims to employ the ΩANTS mechanism also for automated proof construction. Instead of providing suggestions to the user a selector chooses and applies then a suggestion.

In the ΩANTS context, all *inference rules* such as calculus rules, tactics, or methods are uniformly regarded as sets of premises, conclusions, and additional parameters

$$\frac{Prems}{Cons}\ \text{Inference}(Params).$$

The elements of these three sets generally depend on each other. To apply an inference rule at least some of its arguments have to be instantiated by elements of the given proof context, where the arguments that are actually instantiated determine the direction in which the inference rule is applied. The task of the ΩANTS system is now to determine the possible applications of inference rules by computing instantiations for their arguments and to provide the suggestions to the user.

As example consider the calculus rule $\Rightarrow_E \dfrac{P \quad P{\Rightarrow}Q}{Q}$. There are five directions in which this rule can be applied: (i) Forward, 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 as new open line, and (iv) P and Q are given and the implication is introduced as new open line. Finally, closing the subproof, if (v) all three lines are given, then the open goal Q is closed. When applied to a certain proof context, ΩANTS tries to find actual instantiations for the elements of these directions. Thereby ΩANTS first searches for partial instantiations of elements of the five directions that it composes then to complete instantiations. For instance, if ΩANTS finds in the current proof situation a closed line $even(2) \Rightarrow odd(2+1)$ then this is a possible instantiation for $P{\Rightarrow}Q$. This single instantiation pair is already a complete instantiation for direction (ii) and can be part of a complete instantiation for the directions (i), (iii), and (v). If ΩANTS finds also an open line $odd(2+1)$ then it has a complete instantiation for direction (iii). Finally, if it finds a closed line $even(2)$, there is a complete instantiation for direction (v). All complete instantiations are provided as suggestions for the next step to the user. The suggested possibilities are heuristically ordered, for instance, more specific possibilities are preferred before less specific ones. Thus in the discussed example ΩANTS would suggest the instantiations for direction (v), (iii), and (ii) in this order.

Technically, ΩANTS employs a blackboard architecture, that consists of two layers of blackboards: The lower layer of the architecture consists of a set of *rule* *blackboards*, one for each inference rule. We view the knowledge sources of these blackboards as society of agents (i.e., we have one society for each inference rule) since they are realized in independent, concurrent processes. Their task is to search the current $\mathcal{PDS}$ for partial argument instantiations for the inference rule. They communicate via their rule blackboard and can cooperate by adding further specification to a partial argument instantiation other agents have already placed on the blackboard. Each rule blackboard is monitored by one agent that reports the heuristically preferred argument instantiations to the suggestion blackboard, which comprises the upper layer of the architecture. This blackboard accumulates a set of inference rules that are applicable in the current proof state and that are subsequently passed to the user.

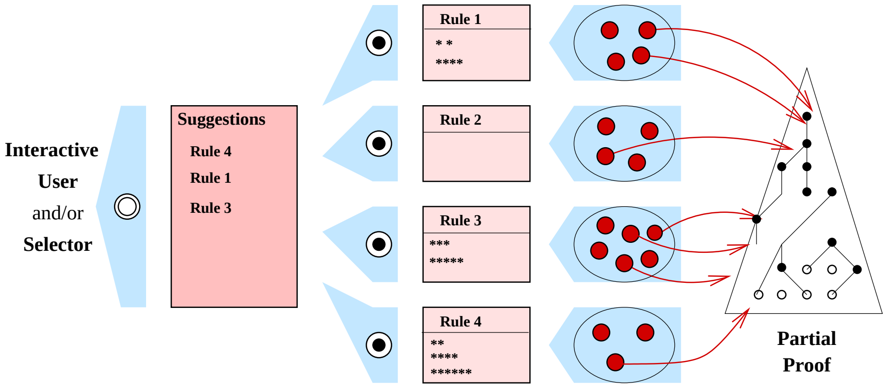


Figure 3.4: The ΩANTS architecture.

A graphical presentation of the ΩANTS architecture is given in Figure 3.4. Agents are displayed by circles, agent societies are grouped in elliptic frames, and blackboards are displayed by boxes. In the figure the architecture is rotated by $\frac{\pi}{2}$; that is, the lower layer with rule blackboards and their respective agent societies are on the right hand side whereas the upper layer with the suggestion blackboard is on the left hand side.

Chapter 4

Knowledge-Based Proof Planning

Proof planning was originally conceived as an extension of tactical theorem proving to enable automated theorem proving at the abstract level of tactics. BUNDY's key idea in [38] is to augment individual tactics with pre- and postconditions. This results in planning operators, so-called *methods*. Thus, proof planning integrates both, elements from tactical theorem proving and elements from AI-planning.

In the ΩMEGA system the traditional proof planning approach is enriched by incorporating mathematical knowledge into the planning process (see [172]). Hence, ΩMEGA's proof planning approach is called *knowledge-based proof planning*. The incorporation of mathematical knowledge is motivated by the observation that mathematicians typically rely on and make use of domain-specific knowledge when proving theorems. In ΩMEGA there are different possibilities to incorporate domain-specific knowledge: in *methods*, in *control rules*, and in *external systems* such as computer algebra systems or constraint solvers. Methods can encode not only 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 in order to guide the search. Moreover, this thesis introduces strategies as further means to incorporate domain knowledge (see chapter 6).

In the remainder of this chapter, we first describe the basics of knowledge-based proof planning, in particular, the languages for methods and control rules and the incorporation of external systems. In the second section, we give a detailed description of ΩMEGA's previous proof planner PLAN to compare it with the new MULTI system later in the thesis. Throughout this chapter we shall relate proof planning to AI-planning. However, we shall give here only a general classification of proof planning with respect to notions from AI-planning. A wider discussion of similarities and differences between proof planning and typical AI-planning can be found in [41, 170, 161].

4.1 Basics of Proof Planning in ΩMEGA

Proof planning in ΩMEGA considers mathematical theorems as planning problems. The *initial state* of a *proof planning problem* consists of the proof *assumptions* and the *goal description* consists of the *theorem*. Methods are the operators of proof planning. A proof planner searches for a solution plan, i.e., a sequence of (instantiated) methods that transforms the initial state into a state in which the theorem holds. In order to find a solution plan, the proof planner searches for applicable methods and applies the instantiated methods. Similar to AI-planning we call the instantiation of a method (i.e., the instantiation of a proof planning operator) an *action*. The effects and the preconditions of an action in ΩMEGA's proof planning are proof lines with formulas in the higher-order language described in section 3.1. The effects of an action should be logically inferable from the preconditions of the action.

Central during the proof planning process are so-called tasks, which express the logical dependencies between goals and assumptions, and a $\mathcal{PDS}$, which represents the partial proof plan constructed so far. We shall now first explain the role of these two fundamental structures.

In AI-planning, an unsatisfied precondition in a plan under construction can be satisfied with a matching effect of any other action in the plan. In proof planning, however, this is not the case because of the logical context of open lines. Thus, ΩMEGA's proof planning uses so-called tasks to express which lines (closed and open) can be used to construct a subplan for an open line. A *task* is a pair $(L_{open}, \mathcal{SUPPS}_{L_{open}})$ where L_{open} is an open line and $\mathcal{SUPPS}_{L_{open}}$ is a set of lines. The first element of a task is called the *task line* or the *goal of the task* and the second element is called the *support lines* or *supports*. The formula of the goal is also called *task formula*. A task with goal L_{open} and supports $\mathcal{SUPPS}_{L_{open}}$ is written as $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$. During the planning process a list of all current tasks is stored in a so-called *agenda*. For a problem with theorem Thm and assumptions $Ass_1, \ldots, Ass_n$ the *initial agenda* consists of the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ where L_{Thm} is an open line with formula Thm and the line L_{Ass_i} has formula Ass_i and is justified with Hyp.

As example for the necessity to maintain a separate set of supports for each goal consider the introduction of a case-split. Let a goal $F[x]$ have the support line $x > 0 \vee x \leq 0$.[1] The introduction of a case-split results in two branches with: subtask $F[x] \blacktriangleleft \{x > 0, \ldots\}$ and $F[x] \blacktriangleleft \{x \leq 0, \ldots\}$. It would be incorrect, if the second subtask used the first assumption or vice versa. Moreover, actions can remove support lines of a task such that afterwards the planner cannot use these lines anymore. This is sensible, for instance, when an action simplifies a given support line with formula $x + 0 > 0$ to the new support with formula $x > 0$. Likely, the old support will not be needed anymore.

The proof plan under construction is represented in a $\mathcal{PDS}$. The *initial* $\mathcal{PDS}$ consists of the lines L_{Thm} and $L_{Ass_1}, \ldots, L_{Ass_n}$. When a new action is added, then the new lines derived by this action are added into the $\mathcal{PDS}$. Moreover, all effect lines of the action are justified by an application of the method of the action to the premises of the action. These applications are tactic applications (since methods are tactics) and are stated in the format described in section 3.2.2. The justifications of the proof lines in the constructed $\mathcal{PDS}$ comprise the same information as causal links known from partial-order planning (see section 2.3): which preconditions of an action are satisfied by which effects of other actions and — vice versa — which effects of an action are used to satisfy which preconditions of other actions. Thus, the $\mathcal{PDS}$ stores information such as which lines are used by actions and which lines depend on which other lines. Moreover, it keeps track of all proof lines created so far. Thereby, open lines in the $\mathcal{PDS}$ represent unsatisfied preconditions of actions (initially, the theorem) whereas closed lines are effects of actions (initially, the proof assumptions).

[1] To simplify this example, we just write the formulas of the goal and the support line instead of the whole proof lines.

During a proof planning process, tasks in the agenda do always correspond to open lines in the $\mathcal{PDS}$, that is, for an open line in the current $\mathcal{PDS}$ there exists a task in the current agenda with this line as goal and vice versa. Thus, with respect to the agenda and the constructed $\mathcal{PDS}$, we can state the aim of the proof planning process as follows: Compute a sequence of actions, which derives, starting from the initial agenda and the initial $\mathcal{PDS}$, an empty agenda and a *closed* $\mathcal{PDS}$, that is, a $\mathcal{PDS}$ without open lines. The *solution proof plan* is a record of this sequence of actions. The simultaneous achievement of an empty agenda and a closed $\mathcal{PDS}$ mirrors the two roots of proof planning: From the AI-planning point of view the aim is to compute a sequence of actions that satisfy all goals, that is, to reach an empty agenda. From the tactical theorem proving point of view the aim is to apply a sequence of tactics, which result in a closed $\mathcal{PDS}$.

The proof planners PLAN and MULTI essentially work on an agenda and its tasks. First, they compute applicable actions for the current tasks. Then, they select one action and apply it. This results in new tasks. Technically, the simultaneous maintenance of a $\mathcal{PDS}$ during the proof planning process is not necessary for the two planners. In particular, if needed, a closed $\mathcal{PDS}$ could be constructed from the computed set of actions later on. However, historically proof planning in ΩMEGA did construct a $\mathcal{PDS}$ and an agenda was only introduced as a bookkeeping mechanism for the open proof lines. Practically, the $\mathcal{PDS}$ is important because of two reasons: First, ΩMEGA's tools for user interaction (e.g., $\mathcal{L}\Omega\mathcal{UI}$) are based on the $\mathcal{PDS}$ as the central data structure. During the proof planning process the constructed $\mathcal{PDS}$ is presented to the user as the current state of progress. When describing the conducted case studies in the chapters 8 — 10 we shall also use $\mathcal{PDS}$s as a means to display and discuss the constructed proof plans. Second, the $\mathcal{PDS}$ is a representation of the current proof plan, i.e., the current sequence of actions, and explicitly stores information that is important for the control rules (e.g., which lines depend on which other lines etc.). Although this information could be computed from the current sequence of actions each time it is needed, it is more convenient to use the $\mathcal{PDS}$ as a bookkeeper.

A formal definition of proof plans and the proof planning process realized in ΩMEGA's previous proof planner PLAN is given in the next section. In the remainder of this section, we introduce ΩMEGA's method and control rule languages, describe actions in ΩMEGA, and briefly discuss the incorporation of external systems into proof planning.

Notation 4.1: Functions that are part of the descriptions of methods, control rules, and algorithms are denoted with a *special font* (e.g., *term-at-position*). Since the core of ΩMEGA is implemented in LISP these functions are LISP functions in the implementation. For clarity, we write the application of the function *func* to the arguments $arg_1, \ldots, arg_2$ not in LISP syntax, i.e., (*func* $arg_1 \ \ldots \ arg_n$), but in prefix notation, i.e., *func*$(arg_1, \ldots, arg_n)$.

Notation 4.2: We denote a *set of items* $it_1, \ldots, it_n$ with $\{it_1, \ldots, it_n\}$. A *list or sequence of items* (i.e., ordered set of items) $it_1, \ldots, it_n$ we write as $[it_1, \ldots, it_n]$. [] denotes the empty list. On sets the operations $\cup, \cap, -$ are defined as usual. On lists $\cup$ denotes the concatenation of lists. The result of $list_1 - list_2$ is $list_1$ without all elements that are in $list_2$. The operations *first*, *last*, *rest*, and *reverse* are defined on lists. The function *first* returns the first element of a list whereas the function *last* returns the last element of a list. The function *rest* returns the list that results from the deletion of the first element from the initial list. The function *reverse* returns a list whose elements are in the reverse order of the elements of the input list.

The set of all items it that satisfy a certain property $P(it)$ is written as $\{it|P(it)\}$.

The analogous list is written as $[it|P(it)]$. The elements of such a list are ordered arbitrarily, if no order is explicitly specified.

Sets are denoted with symbols in calligraphic style (e.g., $\mathcal{M}$ for a set of methods and $\mathcal{C}$ for a set of control rules). Lists are denoted with symbols that are marked with an arrow as superscript (e.g., $\vec{A}$ for a sequence of actions).

4.1.1 Methods

methods encode the knowledge of the relevant proof steps of mathematical domains. Technically, a method in ΩMEGA is a frame data structure with the slots **declarations**, **parameters**, **application conditions**, **premises**, **conclusions**, **outline computations**, **expansion computations**, and **proof schema**.

The *premises and conclusions of a method* specify the preconditions and the effects of the method.[2] The conclusions should be logically inferable from the premises. The union of conclusions and premises is called the *outline* of a method. Declarative descriptions of the formulas of the outline can be given in the proof schema, which also provides the schematic or procedural expansion information (see below).

Premises and conclusions may be annotated with $\oplus$ and $\ominus$. The annotations are needed to indicate whether a method is used for forward or backward search. As opposed to AI-planning, where operators typically can be applied for both forward search and backward search, a method in ΩMEGA is either used in forward search or in backward search. This is because methods typically comprise complex computations that are reasonable either in one direction or in the other direction.

As example, consider methods that employ a computer algebra system to simplify numerical expressions. A backward method can employ the computer algebra system in order to reduce a goal to a simplified goal. A corresponding forward method can employ the computer algebra system in order to derive a simplified support line. But what should the backward method perform when applied forwards? Does it obtain a "simplified" support line and tries to "complicate" it in order to obtain a more "difficult" support? Vice versa, what should the forward method perform when applied backwards? Does it obtain a "simplified" goal, which it tries to "complicate"?

Backward and forward methods are specified as follows: A *backward method* has $\ominus$ *conclusions* and $\oplus$ *premises* as well as $\ominus$ *premises* and *blank premises*. To compute an action of the method, one of the $\ominus$ conclusions is matched with the goal of a given task and both, the $\ominus$ premises and the blank premises, are matched with supports of the task. When the resulting action is introduced into the proof plan, then the goal is closed in the $\mathcal{PDS}$ and the $\oplus$ premises are added to the $\mathcal{PDS}$ and become goals of new tasks. These new tasks inherit the supports of the initial task except that the $\ominus$ premises are removed. The blank premises are not affected. A *forward method* has $\oplus$ *conclusions* as well as $\ominus$ premises and blank premises. To compute an action of the method, the $\ominus$ premises and the blank premises are matched with the support lines of a given task. When the resulting action is introduced into the proof plan, then the $\oplus$ conclusions are added to the $\mathcal{PDS}$ and become new support lines of the task. Moreover, the $\ominus$ premises are removed from the supports of the task. Again, the blank premises are not affected.

[2] That preconditions and effects of a method are called the premises and conclusions of the method, respectively, is an example for the combination of AI-planning and tactical theorem proving in proof planning. If we see the method as tactic, then the effects of a method are the conclusions of a tactic and the preconditions are the premises.

Method: =Subst-B	
declarations	type-variables: α variables: f_o, f'_o, t_α, t'_α, $pos_{position}$ tf_α, tf'_α, $\lambda f_{\alpha o}$
parameters	pos
appl. conds.	(1) *valid-position-p*(f,pos) (2) [*term-at-position*$(f,pos) = t \vee$ *term-at-position*$(f,pos) = t']$
premises	$\oplus L_2$, L_1
conclusions	$\ominus L_3$
outline computations	$f' \leftarrow$ *replace-at-position*(f,t,t',pos)
expansion computations	$tf \leftarrow$ *term-at-position*(f,pos) $tf' \leftarrow$ *term-at-position*(f',pos) $\lambda f \leftarrow$ *lambda-abstraction*(f,pos)
proof schema	$L_1.$ $\vartriangle$ $\vdash t \doteq t'$ $()$ $L_2.$ $\vartriangle$ $\vdash f'$ $(Open)$ $L_4.$ $\vartriangle$ $\vdash \forall P_{\alpha o} \blacksquare P(tf') \Rightarrow P(tf)$ $(\equiv_E \ \doteq)$ $L_5.$ $\vartriangle$ $\vdash (\lambda f)(tf') \Rightarrow (\lambda f)(tf)$ $(\forall_E \ L_4 \ \lambda f)$ $L_6.$ $\vartriangle$ $\vdash f[tf'] \Rightarrow f[tf]$ $(\lambda\leftrightarrow \ L_5)$ $L_3.$ $\vartriangle$ $\vdash f$ $(\Rightarrow_E \ L_2 \ L_6)$

Figure 4.1: The =Subst-B method.

Consider the method =Subst-B, given in Figure 4.1, which can be used in all domains that employ the equality $\doteq$. Essentially, the method performs an equality substitution. It has two preconditions L_1 and L_2, where the proof schema determines L_1 to be an equation. The only conclusion is L_3. =Subst-B is a backward method. The introduction of an action of =Subst-B closes a task line whose formula matches with the formula of L_3 and introduces a new task whose goal is the instantiation of L_2. That is, the formula of the new goal results from the formula of the initial goal by substitution with the equation, which is the formula of a support of the initial task that matched with L_1. For instance, =Subst-B applied to the task $even(a+1) \blacktriangleleft \{a = 1, \ldots\}$[3] introduces the new goal $even(1+1)$.

In the *declarations of a method* the variables of the method and their types are introduced.

The *parameters of a method* are specific variables that influence the resulting action, when the method is instantiated. The =Subst-B method has the parameter *pos* which is of type *position*. The method can be applied to different positions, e.g., for the task $even(a+a) \blacktriangleleft \{a = 1, \ldots\}$ at the first or the second occurrence of a in the goal. The choice of *pos* determines which a should be replaced.

The *application conditions of a method* are meta-level descriptions that restrict the applicability of a method. The application conditions can consist of arbitrary LISP functions. The method =Subst-B has two application conditions: (1) the position *pos* has to be a valid position in the formula f and (2) the subterm in f at the position *pos* is t or t'. Note that application conditions reason only about whether the application of a method is valid in a certain situation; they do not reason about whether the application is useful.

The *outline computations of a method* allow to apply arbitrary LISP functions to compute the new terms and formulas of new outline lines generated by an application of the method. The outline computation of =Subst-B specifies that the

[3] To simplify this example, we just write the formulas of the goal and the support line instead of the whole proof lines.

Method: ∃IResclass-B	
declarations	variables: $c_{\nu o}$, $NSet_{\nu o}$, $RSet_{(\nu o)o}$, P_o, n_ν meta-variables: mv_ν
parameters	
appl. conds.	*resclass-set* $(RSet, n, NSet)$
premises	$\oplus L_3, \oplus L_1$
conclusions	$\ominus L_5$
outline computations	
expansion computations	
proof schema	$L_1.$ △ $\vdash mv \in NSet$ $(Open)$ $L_2.$ △ $\vdash c \in RSet$ $(ConResclSet\ L_1)$ $L_3.$ △ $\vdash P[cl_n(mv)]$ $(Open)$ $L_4.$ △ $\vdash P[c]$ $(ConRescl\ L_3)$ $L_5.$ △ $\vdash \exists x{:}_{RSet}. P[x]$ $(\exists_I Sort\ L_2\ L_4)$

Figure 4.2: The ∃IResclass-B method.

new formula f' is computed from f by replacing t by t' or t' by t at the position *pos* depending on whether the subterm in f at position *pos* is t or t'.

Similarly, the *expansion computations of a method* allow to apply arbitrary LISP functions to compute the new terms and formulas generated during the expansion of an action of the method. The expansion computation of =Subst-B specifies that the terms tf and tf' are computed as the subterms of f and f' at position *pos*, respectively. Moreover, the term λf is computed as a λ-abstraction of f where the term at position *pos* is replaced by the λ -bound variable (that is, essentially λf has the form $\lambda x_\alpha. f[x]$, where $f[x]$ is the term that results from f by replacing the subterm at position *pos* by x).

The *proof schema of a method* is a declarative description of the outline of a method and of the expansion of actions of the method. Expansions of actions corresponds to both tactic expansions and expansions of HTN-planning. When an action of the method is expanded, then for each conclusion a new subproof is introduced into the $\mathcal{PDS}$ resulting in new justifications of the conclusion at a lower level of abstraction. For instance, the proof schema of =Subst-B specifies that the defined concept $\doteq$ in the premise is replaced by its definition (see section 3.2.1). Then, the calculus rules $\forall_E$, $\lambda\leftrightarrow$, and $\Rightarrow_E$ are applied to derive the conclusion of the method.

Another example for a method is ∃IResclass-B given in Figure 4.2, which is a method used for residue class problems (see section 5.2). Its purpose is to instantiate an existentially quantified variable that ranges 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 postpones 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.

∃IResclass-B is a backward method. The introduction of an action of this method reduces a given task whose goal is matched with L_5 to two new tasks whose goals result from L_1 and L_3, respectively. A residue class set is a set of numbers and is annotated by νo (e.g., $c_{\nu o}$). The condition *resclass-set*$(RSet, n, NSet)$ is satisfied if $RSet$, the sort of the quantified variable x, qualifies as a residue class set of the form given in section 5.2. Its evaluation binds the method variables n and $NSet$ to the modulo factor of $RSet$ and the set of integers corresponding to the congruence classes of $RSet$, respectively. For instance, the evaluation of *resclass-set*$(\mathbb{Z}_2, n, NSet)$ yields $n \leftarrow 2$ and $NSet \leftarrow \{0, 1\}$. The necessary inference steps at a lower level of abstraction are indicated by the justifications $ConResclSet$ and $ConRescl$ for the lines L_2 and L_4 in the proof schema, 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.

Notation 4.3: In this thesis, we write mv for meta-variables. If several meta-variables occur, we attach subscripts to mv in order to distinguish the meta-variables. We either use the variable for whose instantiation the meta-variable is a substitute as subscript (e.g., we write mv_x if mv is a substitute for the instantiation of the variable x) or we use numbers. If the decomposition of a quantified formula results in the introduction of a constant, then we write c for this constant. Similar to the notation for meta-variables, we use either the initial variable or numbers as subscripts to distinguish several occurring constants.

Notation 4.4: Methods are written in SMALL CAPITAL FONT (e.g., ∃IRESCLASS-B). The name of backward methods ends with -B whereas the name of forward methods ends with -F.

4.1.2 Actions

An action is an instantiation of a method. Technically, an *action* in ΩMEGA is a frame data structure that has the slots **method**, **task**, **premises**, **conclusions**, **binding**, and **constraints**. The *method of an action* is a pointer to the method of which the action is an instantiation. The *task of an action* is a pointer to the task with respect to which the action was computed. The *conclusions and premises of an action* are sets of proof lines, respectively, which can be annotated with ⊖ and ⊕. The *binding of an action* is a substitution that (1) maps outline lines of the method to proof lines and (2) maps variables specified in the declarations of the method to terms, positions, etc. The *constraints of an action* are constraints that can be created by the evaluation of the application conditions of a method and that have to be passed to external constraint solvers (see section 4.1.4). Similar to methods, we call the union of the premises and conclusions of an action the *outline* of the action. The union of ⊕ premises and ⊕ conclusions of an action is also called the *new lines* of an action (i.e., the proof lines which are produced by an action), whereas the union of ⊖ premises, blank premises, and ⊖ conclusions is called the *given lines* of an action (i.e., the proof lines which have to be given in order to compute an action). Actions of forward methods are also called *forward actions* whereas actions of backward methods are also called *backward actions*.

Example 4.5:

Consider the action in Figure 4.3. It is an instantiation of the method =Subst-B computed with respect to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$. The proof line L_{Thm} is the only conclusion of the action (annotated with ⊖) whereas the proof lines L_{Ass_1} and $L_{Thm'}$ are the premises of the action ($L_{Thm'}$ annotated with ⊕). The binding maps all outline lines of the =Subst-B method (i.e., L_1, L_2, L_3) to the conclusions and the premises of the action and maps all variables declared in =Subst-B to terms and positions. The constraints of this action are empty.

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\ L_{Thm'}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(c+b)\ (Open)$ $L_{Ass_1}.\ L_{Ass_1}\ \vdash\ a \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \rightarrow L_{Thm}, L_1 \rightarrow L_{Ass_1}, L_2 \rightarrow L_{Thm'}, f \rightarrow even(a+b), \alpha \rightarrow \nu,$ $t \rightarrow a, t' \rightarrow c, pos \rightarrow < 1\ 1 >, f' \mapsto even(c+b)\}$
constraints	$\emptyset$

Figure 4.3: An action with the =Subst-B method.

The instantiation of a method in order to compute an admissible action comprises the following steps: First, the formulas of the conclusions and premises have to be matched with formulas of goals and their supports. If this succeeds, then the application conditions can be evaluated. If they evaluate to true, the method is applicable (wrt. to the computed matchings). Then, the outline computations have to be performed and the new lines of the outline have to be computed to complete the action. A detailed description on how actions are computed, selected, and introduced into a proof plan is given in the next section, when we describe PLAN. For the action in Figure 4.3 we give a summary of the computation and introduction into a proof plan here.

Suppose the current $\mathcal{PDS}$ corresponding to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$ is:

$L_{Ass_1}.$	L_{Ass_1}	$\vdash a_\nu \doteq c_\nu$	(Hyp)
$L_{Ass_2}.$	L_{Ass_2}	$\vdash b_\nu \doteq c$	(Hyp)
$L_{Thm}.$	L_{Ass_1}, L_{Ass_2}	$\vdash even_{\nu o}(a+b)$	$(Open)$

When the action in Figure 4.3 is computed, then first the lines L_1 and L_3 of the method =Subst-B are matched with the lines L_{Ass_1} and L_{Thm} of the $\mathcal{PDS}$, respectively. Afterwards, the application conditions are evaluated and the outline computations of the method are performed. Next, the missing outline is computed. In our example, the new $\oplus$ premise $L_{Thm'}$ is computed and is justified with *Open*. When the action is introduced, then its effect L_{Thm} is justified in the $\mathcal{PDS}$ by an application of the method =Subst-B to the premises $L_{Thm'}$ and L_{Ass_1} of the action. Moreover, the new proof line $L_{Thm'}$ is introduced into the $\mathcal{PDS}$. The resulting $\mathcal{PDS}$ is:

$L_{Ass_1}.$	L_{Ass_1}	$\vdash a_\nu \doteq c_\nu$	(Hyp)
$L_{Ass_2}.$	L_{Ass_2}	$\vdash b_\nu \doteq c$	(Hyp)
$L_{Thm'}.$	L_{Ass_1}, L_{Ass_2}	$\vdash even(c+b)$	$(Open)$
$L_{Thm}.$	L_{Ass_1}, L_{Ass_2}	$\vdash even_{\nu o}(a+b)$	(=Subst-B $L_{Thm'}$ L_{Ass_1})

Moreover, the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$ in the agenda is replaced by the task $L_{Thm'} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$.

Proof planning in ΩMEGA is a process that computes actions and introduces them into the proof plan under construction. However, since the introduced actions are represented in the $\mathcal{PDS}$ as applications of their methods we also use the phrase *action application* instead of action introduction, if we want to emphasize the changes in the $\mathcal{PDS}$. We also use the following vocabulary from tactical theorem proving. We say that the application of a backward action *closes an open line or a task*, if the open line or the goal of the task is an effect of the action and is closed by the introduction of the action into the proof plan under construction. We say that a *forward action is applied to some lines or to some supports*, if the lines or supports are the preconditions of the action. Moreover, we say that we *apply a*

```
(control-rule tryanderror-standard-select
              (kind methods)
              (IF (disjunction-supports S))
              (THEN (select (∀IRescalss-B ConCongCl-B
                             ∨E**-B ∃IResclass-B     ))))
```

Figure 4.4: The control rule `tryanderror-standard-select`.

method to a task or to some lines as an abbreviation for the application of an action of the method to the task or to some lines.

4.1.3 Control Rules

Control rules provide guidance of the proof planning process by declaratively representing heuristical knowledge that corresponds to mathematical intuition about how to prove a goal in a certain situation. In particular, these rules provide the basis for meta-level reasoning and a global guidance since they can express conditions for a decision that depends on all available knowledge about the proof planning process so far. Several experiments indicate the superiority of a separate representation of control knowledge by control rules [176]. This representation is well-suited for modifications and for learning. The control rules used in ΩMEGA's proof planning were adopted from the control rule approach of the AI-planner PRODIGY [234],

In the planning process control rules guide decisions at choice points, e.g., which task to tackle next or which method to apply next. They achieve this by reasoning about the heuristic utility of different alternatives[4] in order to promote the alternatives that seem to suit best in the current situation, where 'situation' comprises all available information on the current status such as the current tasks, their supports, the planning history, failed attempts etc. To manipulate an alternative list control rules can remove elements, prefer certain elements, or add new elements. This way, the ranking of alternatives is dynamically changed. This can help to prune the search space or to promote certain promising search paths.

Technically, control rules consist of an IF- and a THEN-part. The IF-part is a predicate on the current proof planning 'situation', whereas in the THEN-part modifications of alternative lists are stated. Moreover, each control rules specifies its kind, i.e., the choice point in the proof planning process it guides.

Figure 4.4 gives as example the control rule `tryanderror-standard-select`, which is evaluated during the selection of the next method to apply. It states that if the current goal is supported by a disjunctive support line S, then the application of the methods ∀IRESCALSS-B, CONCONGCL-B, ∨E**-B, and ∃IRESCLASS-B is attempted in this order.[5] The *select* in the then-part states that all other methods except those specified in the control rule are eliminated from the list of alternative methods. Other possible modifications of alternative lists are *reject*, *prefer*, *defer*, and *order-in-front*. The former removes all alternatives specified in the control rule from a given alternative list, the latter three reorder the alternative list.

[4] As opposed to application conditions of methods, which reason about the legal feasibility of applications of methods (see last section).

[5] ∀IRESCALSS-B and CONCONGCL-B are domain-specific methods to tackle residue class problems where the latter converts statements on residue classes into corresponding statements on integers. The former reduces goals containing a universal quantification over a residue class set similar to ∃IRESCLASS-B. On the contrary, ∨E**-B is not a domain-specific method. It performs a case-split with respect to a set of disjunctive supports.

prefer orders all specified alternatives in front of the alternative list, *defer* orders all specified alternatives at the end of the alternative list, and *order-in-front* orders specified alternatives in front of other specified alternatives. Finally, there is the *insert* modification. It allows to introduce new elements in an alternative list. A typical situation for using an *insert* control rule is when a general control rule – which is applied first – removes some elements from the alternative list, which are needed in a particular situation. Then a more specific *insert* control rule, which is applied later on, can introduce the needed elements again.

Notation 4.6: Control rules are denoted in the `typewriter font` (e.g., `tryanderror-standard-select`). Technically, control rules are frame data structures. Since they are considerably simpler as, for instance, methods, we do not present them in the data structure fashion (as we do with methods) rather we give their LISP encoding. That is, the content of Figure 4.4 is the specification of the control rule `tryanderror-standard-select` as it is in ΩMEGA's data base.

4.1.4 Incorporating External Systems into Proof Planning

We use a special kind of domain knowledge in ΩMEGA, namely the knowledge about and in external "expert" systems. Proof problems usually require many different capabilities for their solution, for instance, computation and object construction. In order to solve problems, it is often necessary to access several systems with complementary capabilities and to make use of their results. Various "expert" systems exist for mathematical problem solving, which have their specific data structures and very efficient algorithms, e.g., computer algebra systems, constraint solvers, model generators, and machine-oriented automated theorem provers. They can support the proof planning process by performing computations, detecting inconsistencies, suggesting instantiations of variables, or solving subproblems. The use of external systems is not just peculiar for proof planning. Rather there are also some AI-planning systems that make use of "experts" [244]. For instance, RAX-PS [125] uses experts in the development of plan fragments.

In general, ΩMEGA's proof planning can treat computations from external systems in two ways: as *hints* or as *proof steps*. The difference is that the soundness of hints is checked by the subsequent proof planning process, which either fails or succeeds for the given hint. To guarantee the soundness of proof steps, special procedures have to be provided, which transform the output of an external system into a subproof that ΩMEGA can check, i.e., special procedures that perform the expansion of such proof steps to ND. Technically, the interface of proof planning to external systems is realized by the LISP functions of methods and control rules. Methods can call external systems in their application conditions and outline computations;[6] similarly, control rules can employ external systems in the predicates of their IF-part.

Figure 4.5 and Figure 4.6 show the two methods COMPLEXESTIMATE-B and TELLCS-B whose application conditions comprise calls to external systems, respectively. Both methods are central for planning limit problems (see section 5.1).

COMPLEXESTIMATE-B is a method for estimating the magnitude of the absolute value of complex terms.[7] COMPLEXESTIMATE-B is applicable to tasks whose goal has the formula $|b| < \epsilon$ (corresponding to line L_9 in Figure 4.5) and that have

[6] Technically, calls of external systems in the expansion computations of methods are also possible. Currently, there is no method that performs such calls.

[7] COMPLEXESTIMATE-B essentially is a reconstruction (see [168]) of BLEDSOE's limit heuristic that was used in a special-purpose program [29].

Method: COMPLEXESTIMATE-B	
declarations	variables: b_ν, ϵ_ν, a_ν, ϵ'_ν, l_ν, k_ν, $a\sigma_\nu$, $k\sigma_\nu$, $l\sigma_\nu$, $b\sigma_\nu$, $\epsilon\sigma_\nu$, $\epsilon'\sigma_\nu$, $conjunct_o$, $\sigma_{substitution}$ meta-variables: mv_ν
parameters	
appl. conds.	*linearextract* (a, b, l, k, σ)
premises	L_1, $\oplus L_2$, $\oplus L_4$, $\oplus L_5$, $\oplus L_6$, $\oplus L_7$
conclusions	$\ominus L_9$
outline computations	$a\sigma :=$ *subst-apply* (σ, a) $k\sigma :=$ *subst-apply* (σ, k) $l\sigma :=$ *subst-apply* (σ, l) $b\sigma :=$ *subst-apply* (σ, b) $\epsilon\sigma :=$ *subst-apply* (σ, ϵ) $\epsilon'\sigma :=$ *subst-apply* (σ, ϵ') $conjunct :=$ *form-conjunction* (σ)
expansion computations	
proof schema	$L_1.$ $\vartriangle$ $\vdash \lvert a \rvert < \epsilon'$ $()$ $L_2.$ $\vartriangle$ $\vdash \epsilon'\sigma < \frac{\epsilon\sigma}{2*mv}$ $(Open)$ $L_3.$ $\vartriangle$ $\vdash \lvert a\sigma \rvert < \frac{\epsilon\sigma}{2*mv}$ $(< trans\ L_1\ L_2)$ $L_4.$ $\vartriangle$ $\vdash \lvert k\sigma \rvert \leq mv$ $(Open)$ $L_5.$ $\vartriangle$ $\vdash \lvert l\sigma \rvert < \frac{\epsilon\sigma}{2}$ $(Open)$ $L_6.$ $\vartriangle$ $\vdash 0 < mv$ $(Open)$ $L_7.$ $\vartriangle$ $\vdash conjunct$ $(Open)$ $L_8.$ $\vartriangle$ $\vdash b\sigma \doteq k\sigma * a\sigma + l\sigma$ (CAS) $L_9.$ $\vartriangle$ $\vdash \lvert b \rvert < \epsilon$ $(fix\ L_3\ L_4\ L_5\ L_6\ L_7\ L_8)$

Figure 4.5: The COMPLEXESTIMATE-B method.

supports with formula $|a| < \epsilon'$ (corresponding to line L_1 in Figure 4.5). In its application conditions COMPLEXESTIMATE-B uses the function *linearextract*. When applied to a and b *linearextract* employs the computer algebra system MAPLE [200] to compute suitable terms k and l such that $b = k * a + l$ holds. *linearextract* also computes a substitution σ such that $b\sigma = k\sigma * a\sigma + l\sigma$ holds (where $b\sigma, k\sigma, l\sigma$ result from b, k, l by the application of the substitution σ, respectively). Thereby, the substitution σ maps meta-variables in a, b to terms. COMPLEXESTIMATE-B is applicable only, if MAPLE provides k and l such that *linearextract* evaluates to true. If this is the case, the application of a corresponding action of the method reduces the original task to five tasks whose goals correspond to the lines L_2, L_4, L_5, L_6, L_7 in Figure 4.5. L_7 has the formula $conjunct$, which is computed from the substitution σ by the function *form-conjunction*. This formula is the conjunction of the mappings of the substitution σ. That is, if σ maps the meta-variables $mv_1, \ldots, mv_n$ to the terms $t_1, \ldots, t_n$, respectively, then $conjunct$ has the form $mv_1 \doteq t1 \wedge \ldots \wedge mv_n \doteq t_n$. If σ is empty, then $conjunct$ is simply $True$, the primitive truth. The justification fix for L_9 in the proof schema is only an abbreviation that stands for a sequence of about 20 tactic steps that comprises, in particular, an application of the triangle inequality. The application of MAPLE is reflected in line L_8 of the proof schema, which is justified by the tactic CAS. When this tactic is expanded, it employs the SAPPER [222] system to obtain a formal proof of the statement $b\sigma = k\sigma * a\sigma + l\sigma$ suggested by MAPLE.

For instance, when applied to a task with formula $|(f(c_x) - g(c_x)) - (l_1 - l_2)| < \epsilon$ and a support with formula $|f(mv_x) - l_1| < \epsilon'$ with a meta-variable

mv_x, then *linearextract* succeeds and provides $k = 1$, $l = g(c_x) - l_2$, and a substitution σ that maps mv_x to c_x. The application of a corresponding action of ComplexEstimate-B reduces the given task to new tasks whose goals are $|1| \leq mv$, $\epsilon' < \frac{\epsilon}{2*mv}$, $|g(c_x) - L_2| < \frac{\epsilon}{2}$, $0 < mv$, and $mv_x \dot{=} c_x$.

Method: TellCS-B	
declarations	variables: a_ν, b_ν, $rel_{\nu\nu o}$
parameters	
appl. conds.	(1) *metavar-in*(a) $\vee$ *metavar-in*(b) (2) *test-CS*$(\mathcal{C}o\mathcal{SIE}, a\ rel\ b)$
premises	
conclusions	$\ominus L_1$
outline computations	
expansion computations	
proof schema	L_1. $\triangle$ $\vdash rel_{o\nu\nu}(a_\nu, b_\nu)$ $(ProveCS)$

Figure 4.6: The TellCS-B method.

The method TellCS-B realizes an interface to $\mathcal{C}o\mathcal{SIE}$ [174], a constraint solver for inequalities and equations over the field of real numbers. TellCS-B is applicable to tasks with formulas $rel_{o\nu\nu}(a_\nu, b_\nu)$ where rel is a binary predicate on arguments of the type ν, which stands for numericals. Examples of matching predicates are, for instance, $<, \leq$. In its application conditions TellCS-B first tests whether a or b contain some meta-variables. If this is the case, $rel(a, b)$ is interpreted as a constraint on these meta-variables. TellCS-B applies then the function *test-CS* that connects to $\mathcal{C}o\mathcal{SIE}$ to test (1) whether $rel(a, b)$ is a syntactically valid constraint for $\mathcal{C}o\mathcal{SIE}$ (in particular, rel has to be $<, \leq, >, \geq, \dot{=}$, or $\neq$) and (2) whether $rel(a, b)$ is consistent with the current constraint store of $\mathcal{C}o\mathcal{SIE}$. If this is the case, TellCS-B is applicable and the corresponding action of TellCS-B contains in its **constraints** slot the constraint $rel(a, b)$. The introduction of the action closes the goal without producing further subtasks and passes $rel(a, b)$ as new constraint to $\mathcal{C}o\mathcal{SIE}$.

Figure 4.7 shows an action of the method TellCS-B. This action contains the constraint $0 < mv_D$, which is annotated with $\mathcal{C}o\mathcal{SIE}$ to indicate that the constraint has to be passed to $\mathcal{C}o\mathcal{SIE}$. The constraint results from the evaluation of the application condition *test-cs* of TellCS-B.

Action	
method	TellCS-B
task	$L_{10} \blacktriangleleft \{L_4, L_5\}$
premises	
conclusions	$\ominus$ L_{10}. $L_4, L_5 \vdash 0 < mv_D$ (*Open*)
binding	$\{L_1 \to L_{10}, a \to 0, b \to mv_D, rel \to <\}$
constraints	$\{\mathcal{C}o\mathcal{SIE}{:}0 < mv_D\}$

Figure 4.7: An action with the TellCS-B method.

$\mathcal{C}o\mathcal{SIE}$ can provide instantiations of the constrained meta-variables that are consistent with the collected constraints. For instance, suppose during the proof planning process there are three tasks whose goals have the formulas $0 < mv_D$, $mv_D < \delta_1$, $mv_D < \delta_2$, which all contain the meta-variable mv_D. All three goals are closed by actions of TellCS-B. Moreover, suppose there are also two supports with formulas $0 < \delta_1$ and $0 < \delta_2$, which are passed to $\mathcal{CoSIE}$ by actions of the method TELLCS-F, which is the analogous of TELLCS-B to pass constraints in supports to $\mathcal{CoSIE}$. From the resulting constraint store, $\mathcal{CoSIE}$ can compute $min(\delta_1, \delta_2)$ as suitable instantiation for mv_D. Moreover, $\mathcal{CoSIE}$ provides traces of its computations, which can be used to expand the applications of the actions of TELLCS-B.

Another method that establishes a connection to $\mathcal{CoSIE}$ is ASKCS-B. Similar to TELLCS-B, this method is applicable to tasks whose goal formulas are of the form $rel(a, b)$. But whereas TELLCS-B demands that a or b contain some meta-variables, ASKCS-B covers the case that a and b contain no meta-variables. An application condition of ASKCS-B passes the formula to $\mathcal{CoSIE}$ and asks $\mathcal{CoSIE}$ whether the formula holds with respect to the constraints collected so far. If this is the case, then ASKCS-B closes the goal. Since $\mathcal{CoSIE}$ can also handle formulas on concrete real numbers, for instance, $1 < 2$ or $0 \leq 0$, ASKCS-B can also close goals whose formulas are expressions on concrete real numbers.

Note that besides TELLCS-B and TELLCS-F also the methods ∀I-B and ∃E-F pass constraints to $\mathcal{CoSIE}$. Actions of ∀I-B perform backward applications of the ND-rule $\forall_I$ by reducing a task with task formula $\forall x_{\bullet} P[x]$ to a new task with task formula $P[c]$, where the variable x is replaced by a constant c. For each meta-variable mv in $P[c]$ an action of ∀I-B also passes the *Eigenvariable constraint* $c!\notin mv$ to $\mathcal{CoSIE}$ that states that the instantiation for mv is not allowed to contain c. This constraint guarantees the adherence with the Eigenvariable conditions of the $\forall_I$ rule of the ND-calculus. Actions of the ∃E-F method perform a forward step with the $\exists_E$ rule. Similar to action of ∀I-B they pass Eigenvariable constraints to $\mathcal{CoSIE}$ that demand the adherence of the Eigenvariable conditions of the $\exists_E$ rule.

4.2 Proof Planning with PLAN

PLAN is ΩMEGA's previous proof planner. It proceeds by successively computing and introducing actions into a proof plan under construction. Preceding the formal description of PLAN (see section 4.2.2), Table 4.1 shows the skeleton of PLAN's algorithm. Essentially, PLAN follows the precondition achievement paradigm (see section 2.3). First, it selects a task to work on. Then, it computes actions for this task and selects one action, which it introduces into the proof plan under construction. This results in new tasks on which PLAN continues. If PLAN fails to compute an action for a selected task, then it performs backtracking. Although actions can perform both, forward reasoning and backward reasoning, an action is always chosen with respect to a task in order to close or to reduce the gap between the goal and the supports of the task.[8] Some decisions in PLAN can be guided by control rules, for instance, the selection of the next task and the selection of the next action. Other decisions, however, are hard-coded into the system. For instance, PLAN employs backtracking if and only if it tackles a task, for which it fails to compute an action. Moreover, it employ external constraint solvers to obtain instantiations for meta-variables if and only if the agenda is empty and the $\mathcal{PDS}$ is closed.

With respect to the notions of AI-planning introduced in section 2.3 we can classify PLAN as follows: PLAN is a state-space planner that combines state-space progression and regression planning. The current progression and regression

[8] In the existing implementation PLAN can introduce a forward action with respect to several tasks simultaneously. This corresponds to the successive application of several actions to a single task, respectively. In order to simplify the formal discussion of PLAN we shall describe the action introduction only with respect to one task.

1. When the current agenda is empty and the current $\mathcal{PDS}$ is closed, then apply external constraint solvers to compute variable instantiations consistent with the collected constraints and terminate.
2. Select a task T from the agenda.
3. Compute and select an action A with respect to T.
4. If an action A could be computed for T, then introduce A. Goto step 1.
5. If no action A could be computed for T, then backtrack the action whose introduction created the task T. Goto step 1.

Table 4.1: Cycle of PLAN.

state are stored in the tasks: the conjunction of all goals is the goal-conjunction of state-space regression planning whereas the union of the supports of the tasks is the current state reached by progression state-space planning. Hence, a node in the search space of PLAN is given by a set of tasks, i.e., an agenda. PLAN starts with the initial agenda. The next node in the search space is reached by the introduction of an action, which changes the agenda etc. A forward action creates a new task by changing the supports of a given task whereas a backward action replaces a task by some new tasks with new goals. The planning process stops as soon as a node in the search space is reached whose set of tasks is empty.

Proof planning does not suffer from the conjunctive goal problems of AI-planners that perform precondition achievement planning. The derivation of a formula F in the subplan for a subgoal is not threatened or removed by the derivation of the negated formula $\neg F$ in the subplan for another subgoal. Hence, PLAN does not perform any threat resolution like demotion or promotion of actions. Moreover, since no re-ordering of introduced actions is performed, PLAN is a total-order planner that computes a sequence of actions.

PLAN's subprocedure for action deletion performs *dependency-directed backtracking* [224]. Instead of backtracking to the last decision point (so-called *chronological backtracking*), the idea of dependency-directed backtracking is to analyze which decisions along a search branch caused a failure. Then, decisions are removed and alternatives are tried based on the found dependencies, rather than the chronological order in which decisions were made. Since there is some ambiguity in the previous use of the term dependency-directed backtracking. We use the term as defined in [202] (p. 212): "*Sometimes, though, we have additional information that tells us which guess (along a search branch) caused the problem. We'd like to retract only that guess and the work that explicitly depended on it, leaving everything else that has happened in the meantime intact. This is exactly what dependency-directed backtracking does.*" Note that in this approach dependency-directed backtracking does not return to an already visited search state but can lead to a new state not visited before. In [100] the same approach is called *dynamic backtracking* because of the dynamic way in which the search is structured. In [127] the term dependency-directed backtracking refers to the approach that analyzes which decision caused a failure and to backtrack to this choice point. That is, all steps done after this decision are removed and an already visited search state is reached again.

Besides the information on the current planning state PLAN has also to maintain information on the search performed so far. In particular, it is necessary to store and make use of information on failing decisions in order to try alternatives instead. Search procedures that perform chronological backtracking often use search trees, which capture possible alternatives as well as made and failed decisions to store information on the traversed search space (e.g., see [1]). Since PLAN performs dependency-directed backtracking we decided for a different approach. PLAN maintains a so-called *history*. A history is a sequence of *manipulation records*. Figure 4.8 shows the skeletons of the two manipulation records, the *action-introduction record* and the *action-deletion record*, of PLAN.

Action-Introduction:	
agenda	
introduced-action	
alternatives	
new-tasks	

Action-Deletion:	
agenda	
deleted-action	

Figure 4.8: Manipulation records in PLAN.

The slot **agenda** captures the context in which the manipulation was done (i.e., the agenda before the manipulation), the slots **introduced-action** and **deleted-action** capture the performed manipulation (i.e., the introduced or deleted action), the slot **alternatives** captures alternative actions available as the introduced action was chosen, and the slot **new-tasks** captures the new tasks created by the application of the chosen action. PLAN records each action introduction or deletion with a corresponding entry in the history. It makes direct use of this information, when selecting the next action: it does not choose again an action that was already deleted (see section 4.2.4). Since PLAN does not return to a particular search state it does not make direct use of the stored alternative actions. However, the information of the history is available to the control rules, which can reason on backtracked steps and possible alternative actions.[9]

In the remainder of this section, we give a detailed description of PLAN. First, we give some formal definitions that culminate in a definition of proof plans and solution proof plans. Then, the subsequent sections give detailed descriptions of PLAN's main algorithm and its subalgorithms for action computation and deletion.

Notation 4.7: In the remainder of the thesis, the following symbols (maybe labeled with some subscripts or superscripts) are associated with the following objects:
$\vec{A}$ denotes a sequence of actions,
$\mathcal{P}$ denotes a $\mathcal{PDS}$,
$\hat{\mathrm{A}}$ denotes an agenda,
$\vec{H}$ denotes a history.

4.2.1 Formal Definition of Proof Plans in PLAN

The aim of this section is to give a formal description of proof plans. We start with definitions of a proof planning problem, an initial $\mathcal{PDS}$ of a proof planning problem, and an initial agenda of a proof planning problem.

Definition 4.8 (Proof Planning Problem): A *proof planning problem* is a quadruple $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{M}, \mathcal{C})$ where Thm and $Ass_1, \ldots, Ass_n$ are formulas in ΩMEGA's higher-order language, $\mathcal{M}$ is a set of methods, and $\mathcal{C}$ is a set of control rules. Thm is also called the *theorem of the proof planning problem* whereas $Ass_1, \ldots, Ass_n$ are called the *assumptions of the proof planning problem*. □

[9] We are currently extending manipulation records to capture also information on the reasons that support a certain decision.

Definition 4.9 (Initial $\mathcal{PDS}$, Initial Agenda): Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{M}, \mathcal{C})$ be a proof planning problem. The *initial* $\mathcal{PDS}$ of this proof planning problem is the $\mathcal{PDS}$ that consists of an open line L_{Thm} with formula Thm and the lines L_{Ass_i} with formula Ass_i and the hypothesis justification Hyp, respectively. The *initial agenda* of the proof planning problem is the agenda that consists of the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$. The task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ is also called the *initial task* of the proof planning problem. □

Next, we define, when an action is applicable with respect to a $\mathcal{PDS}$. Informally speaking, this is the case, when the given lines of the action are in the $\mathcal{PDS}$. Afterwards, we introduce the action introduction function Φ, which describes the operational semantics of an action when it is applied to an agenda, a $\mathcal{PDS}$, and a sequence of actions (i.e., Φ defines a transition relation between triples of agendas, $\mathcal{PDS}$s, and sequences of actions).

Definition 4.10 (Applicable Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$ and A_{add} an action. Moreover, let $\mathcal{L}$ be the set of proof lines of $\mathcal{P}$ and let $\ominus Concs$ be the $\ominus$ conclusions, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}.

A_{add} is *applicable* with respect to $\mathcal{P}$ if

- $(\ominus Concs \;\cup\; \ominus Prems \;\cup\; BPrems)$ is a subset of $\mathcal{L}$.

□

Definition 4.11 (Action Introduction Function Φ): The *action introduction function* Φ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and an action into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\Phi : \vec{A} \times \hat{A} \times \mathcal{P} \times A_{add} \mapsto \vec{A}\text{'} \times \hat{A}\text{'} \times \mathcal{P}\text{'}.$$

Let A_{add} be an action that is applicable with respect to the $\mathcal{PDS}$ $\mathcal{P}$. Let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. Moreover, let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add}.
$Prems$:=$\oplus Prems \cup \ominus Prems \cup BPrems$,
$Concs$:=$\oplus Concs \cup \ominus Concs$
$New\text{-}Lines$:=$\oplus Concs \cup \oplus Prems$
$New\text{-}Supps$:=$(\mathcal{SUPPS}_{L_{open}} \cup \oplus Concs) - \ominus Prems$.
$New\text{-}Tasks$:=$[L \blacktriangleleft New\text{-}Supps \mid L \in \oplus Prems]$.

If $\vec{A}$ is a sequence of actions and $\hat{A}$ is an agenda that contains the task T of A_{add}, then the result $(\vec{A}\text{'}, \hat{A}\text{'}, \mathcal{P}\text{'})$ of $\Phi(\vec{A}, \hat{A}, \mathcal{P}, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- $\hat{A}\text{'}:= \begin{cases} New\text{-}Tasks \cup (\hat{A} - [T]) & \text{if } L_{open} \in \ominus Concs, \\ [L_{open} \blacktriangleleft New\text{-}Supps] \cup New\text{-}Tasks \cup (\hat{A} - [T]) & \text{else.} \end{cases}$
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. adding the proof lines $New\text{-}Lines$, respectively, and
  2. justifying the proof lines $\ominus Concs$ and $\oplus Concs$ with the justification $(M\ Prems)$, respectively, where M is the method of A_{add}.

□

The recursive extension Φ is called $\vec{\Phi}$. $\vec{\Phi}$ introduces a whole sequence of actions (the arrow of $\vec{\Phi}$ indicates that this function introduces a sequence of actions $\vec{A}_{add}$).

Definition 4.12 (Recursive Action Introduction Function $\vec{\Phi}$): The *recursive action introduction function* $\vec{\Phi}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of actions into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\vec{\Phi} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times \vec{A}_{add} \mapsto \vec{A}\text{'} \times \hat{\mathrm{A}}\text{'} \times \mathcal{P}\text{'}.$$

$\vec{\Phi}$ is recursively defined as follows:
Let $\vec{A}$ be a sequence of actions, $\hat{\mathrm{A}}$ an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, and $\vec{A}_{add}$ a sequence of actions.

1. If $\vec{A}_{add}$ is empty then $\vec{\Phi}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{A}_{add}) = (\vec{A}, \hat{\mathrm{A}}, \mathcal{P})$.
2. Otherwise let $A_{add} :=$ *first*$(\vec{A}_{add})$ and $\vec{A}\text{'}_{add} :=$ *rest*$(\vec{A}_{add})$. If A_{add} is applicable with respect to $\mathcal{P}$, and if $\hat{\mathrm{A}}$ contains the task of A_{add}, then
$$\vec{\Phi}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{A}_{add}) = \vec{\Phi}(\Phi(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, A_{add}), \vec{A}\text{'}_{add}).$$

□

With the function $\vec{\Phi}$ we can now define proof plans and solution proof plans.

Definition 4.13 (Proof Plans and Solution Proof Plans):
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{M}, \mathcal{C})$ be a proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and $\hat{\mathrm{A}}_{init}$ its initial agenda.

A *proof plan* for the proof planning problem is a triple $\mathsf{PP} = (\vec{A}, \hat{\mathrm{A}}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda $\hat{\mathrm{A}}$, and a $\mathcal{PDS}$ $\mathcal{P}$ such that:

1. the methods of each action of $\vec{A}$ are in $\mathcal{M}$,
2. $(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}) = \vec{\Phi}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, \vec{A})$,

A *solution proof plan* for the proof planning problem is a sequence of actions $\vec{A}$ such that $\vec{\Phi}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, \vec{A})$ has an empty agenda and a closed $\mathcal{PDS}$. □

Because of this definition, we can also say that Φ maps a proof plan and an action into a proof plan and that $\vec{\Phi}$ maps a proof plan and a sequence of actions into a proof plan.

4.2.2 The PLAN Algorithm

Figure 4.9 gives a pseudo-code description of the PLAN algorithm. PLAN obtains as input a proof plan $\mathsf{PP} = (\vec{A}, \hat{\mathrm{A}}, \mathcal{P})$, a history $\vec{H}$, a list of methods $\mathcal{M}$, and a list of control rules $\mathcal{C}$.[10] PLAN generates a sequence of pairs of proof plans PP and histories $\vec{H}$. The user of ΩMEGA can start PLAN with the initial $\mathcal{PDS}$, the initial agenda, and the set of methods and control rules of a proof planning problem. In order to reach the next proof plan and the next history PLAN performs a cycle of termination check, task selection, action selection and action introduction or action deletion. It terminates when either the agenda of the current proof plan is empty

[10] Both methods $\mathcal{M}$ and control rules $\mathcal{C}$ are lists and not sets since the order in these lists are relevant. The order in $\mathcal{M}$ gives a default order in which the methods are tried, when no control rules fire and determine a different order (see section 4.2.4). The order in $\mathcal{C}$ determines the order in which the control rules are evaluated.

Input: (1) a proof plan PP $= (\vec{A}, \hat{A}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda $\hat{A}$, and a $\mathcal{PDS}$ $\mathcal{P}$, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a solution proof plan and a closed $\mathcal{PDS}$ or **fail**.

```
Algorithm: PLAN(($\vec{A}$, $\hat{A}$, $\mathcal{P}$),$\vec{H}$,$\mathcal{M}$,$\mathcal{C}$)

  1. Termination
     If $\hat{A}$ is empty, then terminate and return employ-CS($\vec{A}$,$\mathcal{P}$).

  2. Task Selection
     Let current task $T$:= first(evalcrules-tasks($\hat{A}$,$\mathcal{C}$))
     where $T$ is the pair $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

  3. Action Selection
     Let ($A_{add}$,$\mathcal{A}$):=CHOOSEACTION($T$,$\vec{H}$,$\mathcal{M}$,$\mathcal{C}$)
     where $A_{add}$ is an action and $\mathcal{A}$ is a set of alternative actions.

  4. Action Introduction
     If $A_{add}$ is given
       then
         ($\vec{A}$',$\hat{A}$',$\mathcal{P}$'):=$\Phi(\vec{A}, \hat{A}, \mathcal{P}, A_{add})$.
         $\vec{H}$':=add-action-intro-record($\vec{H}$,$\hat{A}$,$A_{add}$,$\mathcal{A}$).
         If extract-constraints($A_{add}$) $\neq \emptyset$
           then
             pass-constraints(extract-constraints($A_{add}$)).
         PLAN(($\vec{A}$',$\hat{A}$',$\mathcal{P}$'),$\vec{H}$',$\mathcal{M}$,$\mathcal{C}$).

  5. Action Deletion
     If $A_{add}$ is not given
       then
         If $\vec{A}$ is empty
           then
             Terminate and return fail.
           else
             Let $A_{reason}$:=find-introducing-action(T,$\vec{H}$).
             (($\vec{A}$',$\hat{A}$',$\mathcal{P}$'),$\vec{H}$'):=BACKTRACK(($\vec{A}$,$\hat{A}$,$\mathcal{P}$),$\vec{H}$,[$A_{reason}$]).
             PLAN(($\vec{A}$',$\hat{A}$',$\mathcal{P}$'),$\vec{H}$',$\mathcal{M}$,$\mathcal{C}$).
```

Figure 4.9: The PLAN algorithm.

(see step 1 in Figure 4.9) or when there are neither further actions to be introduced nor actions to be removed (see step 5 in Figure 4.9). In the former case PLAN was successful and returns the proof plan and the constructed closed $\mathcal{PDS}$. In the latter case, PLAN did traverse the complete search space without finding a proof plan and returns **fail**.

If the current agenda is not empty, then PLAN first selects the next task to tackle (step 2 in Figure 4.9). To do so, PLAN employs the function *evalcrules-tasks*. *evalcrules-tasks* evaluates the control rules $\mathcal{C}$ of the kind 'Tasks' on the tasks list of the current agenda and returns a (possibly) changed alternative list.[11] Then,

[11] Although we do not explicitly provide the current proof plan and the current history as

PLAN picks the fist element of the resulting list as current task.

Next, PLAN employs the subalgorithm **ChooseAction** to compute an action (step 3 in Figure 4.9). **ChooseAction** is applied to the current task, the methods $\mathcal{M}$, and the control rules $\mathcal{C}$. It tries to compute admissible actions and – if successful – it selects one action and returns it. Since **ChooseAction** is a complex algorithm we shall discuss it in detail in section 4.2.4.

If **ChooseAction** returns an action, then PLAN introduces the action (step 4 in Figure 4.9). It creates a new proof plan by applying the action introduction function Φ to the current proof plan and the chosen action. Moreover, it creates a new history by adding a new action-introduction record entry to the history. PLAN uses the function *extract-constraints* to access the constraints of an action. When the action contains constraints for the connected external constraint solvers, then PLAN employs the function *pass-constraints*, which passes the constraints to the respective external system. PLAN does not check whether the new constraints are accepted by the respective external system. Rather, it assumes that corresponding consistency checks are performed by **ChooseAction** as part of the evaluation of the application conditions of a method, when an action is computed.

When **ChooseAction** fails to provide an action, then PLAN tries to delete actions in the current proof plan (step 5 in Figure 4.9). If the current sequence of actions is empty, then this is obviously not possible. When there are no more actions that can be introduced and the current sequence of actions is empty, then PLAN did traverse the complete search space (complete wrt. to the methods $\mathcal{M}$ and the control rules $\mathcal{C}$) without finding a solution proof plan. In this case, PLAN terminates and returns **fail**. If there are actions that can be deleted, then PLAN employs the function *find-introducing-action* to determine the action whose introduction created the task T for which no action can be computed. The information about which action introduction did introduce which task can be found in the history in the action-introduction entries. Then, PLAN employs the subalgorithm **Backtrack** to perform the deletion of the selected action and all further actions that explicitly depend on it. **Backtrack** is applied to the current proof plan, the current history, and a list with the action to be deleted as only element. It returns a changed proof plan and a changed history. Since **Backtrack** is a complex algorithm we shall discuss it in detail in the next section.

When the agenda is empty, then the introduction of actions stops and PLAN applies the function *employ-CS* to the computed action sequence and the constructed $\mathcal{PDS}$ (step 1 in Figure 4.9). This function employs the external constraint solvers to compute instantiations for the meta-variables. Then, it substitutes all occurrences of the meta-variables in proof lines of the $\mathcal{PDS}$ and the actions by their instantiations, respectively. It returns the resulting action sequence and the instantiated $\mathcal{PDS}$, which are then the output of PLAN.

Although proof planning actions are complex actions in the sense of HTN-planning, the expansion of actions is not performed within PLAN. Rather, there are separate procedures in ΩMEGA for the expansion of actions. When an expansion fails to produce a calculus-level proof and results in new open lines, then PLAN can be re-invoked on the new tasks.

arguments for *evalcrules-tasks*, the predicates in the IF-part of the evaluated control rules can make use of this status information. This holds for all kinds of control rules, not only for the control rules of kind ‘Tasks’ evaluated here.

4.2.3 Deletion of Actions

Before we describe the **BACKTRACK** algorithm, we shall introduce the notion of dependency among actions and when an action is deletable. When an action is introduced into a proof plan, then it modifies the elements of the proof plan. Other actions introduced later on may depend on these modifications. More concretely, when the new lines introduced by an action are used as given lines by other actions introduced later on, then these actions depend on the preceding action. Afterwards, we define the function for the deletion of an action from a proof plan. Since action deletion is conceptually the inverse operation of action introduction we call this function Φ^{-1} although technically Φ^{-1} is not the inverse function of Φ.

Definition 4.14 (Dependent Actions): Let $\vec{A}$ be a sequence of actions with $\vec{A}$=$[A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n]$. Let A_i be an action with the $\oplus$ conclusions $\oplus$*Concs*, and the $\oplus$ premises $\oplus$*Prems*. An action $A_j \in \{A_{i+1},\ldots,A_n\}$ depends on A_i, if A_j is an action whose sets of conclusions or premises contains a proof line of $\oplus$*Concs* or $\oplus$*Prems* (which are the new proof lines introduced by A_i). □

Definition 4.15 (Deletable Actions): Let $\vec{A}$ be a sequence of actions with $\vec{A}$=$[A_1,\ldots,A_{i-1},A_{del},A_{i+1},\ldots,A_n]$. A_{del} is *deletable* with respect to $\vec{A}$, if the set of actions in $\vec{A}$ that depend on A_{del} is empty. □

In the following definition of the function Φ^{-1} we describe the modifications of the sequence of actions, the agenda, and the $\mathcal{PDS}$ caused by the deletion of an action. Although the notion of deletability of an action is defined only with respect to a sequence of actions, we demand in the definition of Φ^{-1} that the agenda and the $\mathcal{PDS}$ are not arbitrary ones, but created by this sequence of actions (in particular, by the action that should be deleted). The described modifications cannot be performed with respect to an arbitrary $\mathcal{PDS}$ or an arbitrary agenda.

Definition 4.16 (Action Deletion Function Φ^{-1}): The *action deletion function* Φ^{-1} is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, and an action into a sequence of actions, an agenda, and a $\mathcal{PDS}$, i.e.,

$$\Phi^{-1} : \vec{A} \times \hat{A} \times \mathcal{P} \times A_{del} \mapsto \vec{A}\text{'} \times \hat{A}\text{'} \times \mathcal{P}\text{'}.$$

Let A_{del} be a deletable action in $\vec{A}$. Let $\oplus$*Concs* be the $\oplus$ conclusions, $\ominus$*Concs* the $\ominus$ conclusions, $\oplus$*Prems* the $\oplus$ premises, $\ominus$*Prems* the $\ominus$ premises, and *BPrems* the blank premises of A_{del}. Moreover, let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del}.
Lines-To-Remove:=$\oplus$*Concs* $\cup$ $\oplus$*Prems*.
Tasks-To-Remove:=$[L \blacktriangleleft \mathcal{SUPPS}_L \in \hat{A} \mid L \in \oplus Prems]$.
New-Tasks:=$[T]$.

If $\hat{A}$ is an agenda and $\mathcal{P}$ is a $\mathcal{PDS}$ that results from the introduction of $\vec{A}$ (to some agenda and some $\mathcal{PDS}$), then the result $(\vec{A}\text{'},\hat{A}\text{'},\mathcal{P}\text{'})$ of $\Phi^{-1}(\vec{A},\hat{A},\mathcal{P},A_{del})$ is defined by:

- $\vec{A}$':= $\vec{A} - [A_{del}]$.
- $\hat{A}$':= *New-Tasks* $\cup$ ($\hat{A}$ $-$ *Tasks-To-Remove*).
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. removing the lines *Lines-To-Remove* and
  2. justifying the proof lines $\ominus$*Concs* with *Open*, respectively.

□

```
Input: (1) a proof plan PP = $(\vec{A}, \hat{A}, \mathcal{P})$ with a sequence of actions $\vec{A}$, an agenda $\hat{A}$, and a $\mathcal{PDS}$
       $\mathcal{P}$, (2) a history $\vec{H}$, (3) a sequence of actions $\vec{A}_{del}$.

Output: A proof plan PP' = $(\vec{A}', \hat{A}', \mathcal{P}')$ and a history $\vec{H}'$.

Algorithm: Backtrack($(\vec{A}, \hat{A}, \mathcal{P})$,$\vec{H}$,$\vec{A}_{del}$)

  1. Termination
     If $\vec{A}_{del}$ is empty, then terminate and return $((\vec{A}, \hat{A}, \mathcal{P}), \vec{H})$.

  2. Pick Action
     Let $A_{del}$:=first($\vec{A}_{del}$).

  3. Action Deletion
     If $A_{del}$ is deletable wrt. $\vec{A}$
       then
         $(\vec{A}', \hat{A}', \mathcal{P}')$:=$\Phi^{-1}(\vec{A}, \hat{A}, \mathcal{P}, A_{del})$.
         $\vec{H}'$:=add-action-del-record($\vec{H}$,$\hat{A}$,$A_{del}$).
         If extract-constraints($A_{del}$) $\neq \emptyset$
           then
             delete-constraints(extract-constraints($A_{del}$)).
         BACKTRACK($(\vec{A}', \hat{A}', \mathcal{P}')$,$\vec{H}'$,rest($\vec{A}_{del}$)).

  4. Deletion Expansion
     If $A_{del}$ is not deletable wrt. $\vec{A}$
       then
         $\vec{A}_{del}^{new}$:=dependend-actions($A_{del}$, $\vec{A}$).
         BACKTRACK($(\vec{A}, \hat{A}, \mathcal{P})$,$\vec{H}$,$\vec{A}_{del}^{new} \cup \vec{A}_{del}$).
```

Figure 4.10: The **BACKTRACK** algorithm.

A pseudo-code description of the algorithm **BACKTRACK** is given in Figure 4.10. **BACKTRACK** is applied to a proof plan PP $= (\vec{A}, \hat{A}, \mathcal{P})$, a history $\vec{H}$, and a list of actions $\vec{A}_{del}$ that have to be deleted. **BACKTRACK** generates a sequence of pairs of proof plans PP and histories $\vec{H}$ by deleting successively the actions in $\vec{A}_{del}$. If an action in $\vec{A}_{del}$ is not deletable, then it is necessary to delete further actions. **BACKTRACK** returns the proof plan and the history that result from the deletion of all necessary actions.

The first step in **BACKTRACK** is a check whether the list of actions that should be deleted is empty. If this is the case, **BACKTRACK** terminates and returns the current proof plan and the current history. Otherwise, it selects the first action A_{del} from the list (step 2 in Figure 4.10). If A_{del} is deletable, **BACKTRACK** deletes it from the current proof plan by employing Φ^{-1} and adds a new action-deletion entry to the history (step 3 in Figure 4.10). When A_{del} contains constraints, then **BACKTRACK** employs the function *delete-constraints*, which tells the respective constraint solvers to delete these constraints since they are not longer existing. Afterwards, **BACKTRACK** is applied to the changed proof plan, the changed history, and the remaining actions to be deleted.

If A_{del} is not deletable (step 4 in Figure 4.10), then **BACKTRACK** calls the function *dependent-actions* to compute the actions that depend from A_{del} and that have to be deleted in order to make A_{del} deletable. **BACKTRACK** is then recursively applied to the current proof plan, the current history, and the concatenation of the actions computed by *dependent-actions* and the current actions that have to be deleted.

As example for a situation, where an action is not deletable because other actions depend on it, consider the following situation. PLAN introduces an action A that reduces a task with goal L to two new tasks with goals L_1 and L_2. Next, PLAN applies the action A_1 to close L_1. Afterwards, PLAN fails to apply an action to the task with goal L_2 and employs **BACKTRACK** to remove the action A that introduced L_2. However, the deletion of A would not only remove the line L_2 but also the line L_1 with respect to which action A_1 was introduced. Hence, before A can be deleted the action A_1 has to be deleted.

4.2.4 Action Computation and Selection

CHOOSEACTION is the subalgorithm of PLAN that computes alternative lists of actions and selects one of them. Figure 4.11 shows a pseudo-code description of the algorithm. **CHOOSEACTION** is applied to a task, the current history, and the lists of methods $\mathcal{M}$ and control rules $\mathcal{C}$. If successful, **CHOOSEACTION** returns a selected action and a set of alternative actions (see step 7 in Figure 4.11), otherwise it returns **fail** (see step 2 in Figure 4.11).

CHOOSEACTION computes actions successively. It starts with an under-specified, initial action that contains only a chosen method and the given task. Then, it successively matches lines of the method with the goal and the supports of the task as well as variables specified in the declarations of the method with terms, positions, etc. The substitutions of these matchings refine successively the binding of the action such that more and more specified actions are created. In order to check whether a particular action of a method is valid, **CHOOSEACTION** evaluates the application conditions of the method with respect to the binding of the action. Afterwards, it completes the binding of the actions by conducting the outline computations and by computing the new lines. Finally, it selects one action among the resulting fully specified actions.

In the following, we explain **CHOOSEACTION** with the example 4.5 of section 4.1.2. We apply **CHOOSEACTION** to the task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$, an empty history, a list of methods that contains the method =Subst-B, and a list of control rules that contains the control rule `supps+params-=Subst` whose impact is explained below.

The first step in **CHOOSEACTION** is the re-ordering of the alternative list of methods. This is done by the function *evalcrules-methods*, which obtains as input $\mathcal{M}$, $\mathcal{C}$ and the given task. *evalcrules-methods* evaluates the control rules in $\mathcal{C}$ of kind 'Methods' on $\mathcal{M}$ and returns a (possibly) changed list of alternative methods. From this list **CHOOSEACTION** picks the first one (step 2 in Figure 4.11) and employs the function *initial-action-set* to create the initial set of actions that consists of one action whose premises, conclusions, bindings, and constraints are empty, whose method is the chosen method, and whose task is the given task.

For our example, we assume that *evalcrules-methods* returns the list [=Subst-B, ...]. Then, **CHOOSEACTION** chooses =Subst-B as method and produces an initial set of actions that contains only the following action:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	
conclusions	
binding	
constraints	

Input: (1) a task T, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a pair of an action and a list of actions or **fail**.

```
Algorithm: ChooseAction($T$,$\vec{H}$,M,C)
Let $T{=}L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

  1. Order Methods
     $Methods$:= evalcrules-methods($\mathcal{M}$,$\mathcal{C}$,$T$).

  2. Select Method
     If $Methods$ empty
        then
            Terminate and return fail.
        else
            $M$:=first($Methods$).
            $Actions$:=initial-action-set($T, M$).

  3. Match Goal
     Let $\ominus Concs$ be the $\ominus$ conclusions of $M$.
     $Actions$:=match-goal($L_{open}$,$\ominus Concs$,$Actions$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  4. Select and Match Supports and Parameters
     Let $\ominus Prems$ and $BPrems$ be the $\ominus$ premises and blank premises of $M$.
     Let $Params$ be the parameter variables of $M$.
     $Supps{+}Params$:=evalcrules-s+p($\mathcal{SUPPS}_{L_{open}}$,$\mathcal{C}$,$T$,$M$,$Actions$).
     $Actions$:= match-s+p($Supps{+}Params$,$\ominus Prems \cup BPrems$,
                                                        $Params$,$Actions$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  5. Evaluate Application Conditions
     $Actions$:=eval-appl-conds($Actions$,$M$).
     If $Actions$ empty, then $Methods$:=rest($Methods$), goto 2.

  6. Outline Computations
     eval-outline-computations($Actions$).
     complete-outline($Actions$).

  7. Select an Action
     $Actions$:=remove-backtracked($Actions$,$\vec{H}$).
     $Actions$:=evalcrules-actions($Actions$,$\mathcal{C}$).
     If $Actions = \emptyset$
        then
            $Methods$:=rest($Methods$), goto 2.
        else
            Terminate and return (first($Actions$),rest($Actions$)).
```

Figure 4.11: The **ChooseAction** algorithm.

The next step (step 3 in Figure 4.11) in **ChooseAction** matches the goal with the $\ominus$ conclusions of the selected method. To do so, **ChooseAction** employs the function *match-goal*. This function is applied to the goal, the $\ominus$ conclusions of the selected method, and the set of actions computed so far. Its computations and its output depend on the existence of $\ominus$ conclusions in the chosen method. If the method has no $\ominus$ conclusions (i.e., a forward method), then *match-goal* simply returns the list of actions it obtained as input. If the method has $\ominus$ conclusions (i.e., a backward method), then *match-goal* matches the goal with the $\ominus$ conclusions, respectively. For each successful matching it creates a new action whose binding contains the substitution resulting from the matching and whose conclusions contain the goal annotated with $\ominus$. Finally, *match-goal* returns the set of all new actions.

In our example the matching of the goal L_{Thm} with the $\ominus$ conclusions of =Subst-B results in a substitution with two elements: $L_3 \mapsto L_{Thm}$ and $f \mapsto even(a+b)$. Thus, *match-goal* returns an actions set that contains only the following action:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, f \mapsto even(a+b)\}$
constraints	

Next, ChooseAction chooses supports and parameters and matches them with $\ominus$ and blank premises and the parameter variables of the selected method (step 4 in Figure 4.11). This results in further substitutions, which refine the actions computed so far. First, ChooseAction evaluates the control rules of the kind 'Supps+Params'. This is done by the function *evalcrules-s+p*, which is applied to the supports of the goal, the control rules $\mathcal{C}$, the task, the current method, and the actions computed so far. Control rules of the kind 'Supps+Params' do not only reorder and manipulate the support lines but they return a new type of elements, namely pairs of support lines and parameter instantiations. Thus, the parameter selection is not an isolated decision but is combined with the selection of support lines.[12] Then, ChooseAction employs the function *match-s+p*. *match-s+p* obtains as input the pairs of support lines and parameter instantiations, the $\ominus$ and blank premises of the selected method, and the set of actions computed so far. With respect to each action computed so far (i.e., depending on the binding of an action computed so far) *match-s+p* matches the support lines and parameters pairs with the $\ominus$ and blank premises and the parameter variables of the method, respectively. For each successful matching it creates a new action whose binding is extended with the substitution resulting from the matching and whose premises comprise the matched support lines. Finally, *match-s+p* returns the set of new actions.

In our example, the control rule `supps+params-=Subst` fires and returns the two support lines and parameter instantiation pairs $(\{L_{Ass_1}\}, <1\ 1>)$ and $(\{L_{Ass_2}\}, <1\ 2>)$, where $<1\ 1>$ is the parameter position of the a in the formula $even(a+b)$ of the goal L_{Thm} and $<1\ 2>$ is the parameter position of the b.[13] For both pairs and with respect to the only action computed so far, *match-s+p* succeeds to match the premise L_1 and the parameter *pos* of =Subst-B with the content of the pairs, respectively. It returns a set of actions that contains the following two elements:

[12] We decided for this combined approach since typically the parameter selection is directly related to the support line selection.

[13] The control rule `supps+params-=Subst` fires if the current method is =Subst-B and if there are some support lines that are equations such that one side of the equations equals a subterm in the formula of the goal. If `supps+params-=Subst` finds such a support line it returns a pair consisting of the support line and the respective subterm position in the formula of the goal.

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$L_{Ass_1}.\ L_{Ass_1} \ \vdash\ a \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \rightarrow L_{Ass_1}, f \mapsto even(a+b), \alpha \rightarrow \nu,$ $t \rightarrow a, t' \rightarrow c, pos \rightarrow < 1\ 1 >\}$
constraints	

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$L_{Ass_2}.\ L_{Ass_2} \ \vdash\ b \doteq c\ (Hyp)$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2} \ \vdash\ even(a+b)\ (Open)$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \rightarrow L_{Ass_2}, f \mapsto even(a+b), \alpha \rightarrow \nu,$ $t \rightarrow b, t' \rightarrow c, pos \rightarrow < 1\ 2 >\}$
constraints	

The first action results from matching L_1 and pos with L_{Ass_1} and $< 1\ 1 >$, respectively, whereas the second action results from matching L_1 and pos with L_{Ass_2} and $< 1\ 2 >$, respectively.

In the next step (step 5 in Figure 4.11), **ChooseAction** evaluates the application conditions of the selected method. The evaluation of the application conditions is performed by the function *eval-appl-conds*, which obtains as input the actions computed so far and the selected method. For each given action *eval-appl-conds* evaluates the application conditions of the method with respect to the binding of the action. The evaluation of application conditions can create further substitutions, which are then added to the binding of the action. Moreover, the evaluation can create constraints for external constraint solvers, which are then added as constraints of the action. Each action for which the evaluation fails is rejected. *eval-appl-conds* returns the set of all actions for which the evaluation succeeds.

In our example, the application conditions of =Subst-B evaluate to **true** for both actions computed so far. Since no constraint results from the evaluation of the application conditions the constraints of both actions are set to the empty set.

Next, **ChooseAction** completes the actions by conducting the outline computations of the selected method and by computing the new outline lines (i.e., $\oplus$ premises and conclusions) (see step 6 in Figure 4.11). This is done by the functions *eval-outline-computations* and *complete-outline*, which both are applied to the set of actions computed so far. Both functions do not change the set of actions but they refine the actions already in the set. *eval-outline-computations* evaluates the outline computations for each action and adds the resulting substitutions to the binding of the action. Similarly, *complete-outline* computes the missing outline lines for each action and adds the corresponding substitutions to the binding of the action. New outline lines are justified as follows: $\oplus$ premises are justified with *Open* whereas new $\oplus$ conclusions are justified by an application of the selected method to the premises of the action.

For our example, *eval-outline-computations* and *complete-outline* complete the actions computed so far as follows:

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\ L_{Thm'}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ \mathit{even}(c+b)\ (\mathit{Open})$ $L_{Ass_1}.\ L_{Ass_1}\ \vdash\ a \doteq c\ (\mathit{Hyp})$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ \mathit{even}(a+b)\ (\mathit{Open})$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \rightarrow L_{Ass_1}, L_2 \rightarrow L_{Thm'}, f \mapsto \mathit{even}(a+b), \alpha \rightarrow \nu,$ $t \rightarrow a, t' \rightarrow c, \mathit{pos} \rightarrow < 1\ 1 >, f' \rightarrow \mathit{even}(c+b)\}$
constraints	$\emptyset$

Action	
method	=Subst-B
task	$L_{Thm} \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
premises	$\oplus\ L_{Thm'}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ \mathit{even}(a+c)\ (\mathit{Open})$ $L_{Ass_2}.\ L_{Ass_2}\ \vdash\ b \doteq c\ (\mathit{Hyp})$
conclusions	$\ominus\ L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash\ \mathit{even}(a+b)\ (\mathit{Open})$
binding	$\{L_3 \mapsto L_{Thm}, L_1 \rightarrow L_{Ass_2}, L_2 \rightarrow L_{Thm'}, f \mapsto \mathit{even}(a+b), \alpha \rightarrow \nu,$ $t \rightarrow b, t' \rightarrow c, \mathit{pos} \rightarrow < 1\ 2 >, f' \rightarrow \mathit{even}(a+c)\}$
constraints	$\emptyset$

Finally, **ChooseAction** decides for one of the computed actions (step 7 in Figure 4.11). First, it rejects all actions that correspond to actions that have already been backtracked. This is done by the function *remove-backtracked*, which is applied to the current set of actions and the given history. If an action has the same given lines and the same binding as an action that is stored in the history as deleted action, then this action is removed from the alternative list. To the remaining actions **ChooseAction** applies the function *evalcrules-actions* to evaluate the control rules of kind 'Actions'. Provided the resulting list of actions is not empty, **ChooseAction** terminates and returns a pair consisting of the first element of the list of actions and the rest of the list of actions (i.e., the chosen action and the list of alternatives). If the list of actions is empty, then **ChooseAction** returns to the method selection point (step 2 in Figure 4.11) and repeats the sequence of matchings, application condition evaluation, outline computations evaluation, and outline completion for the next method of the method list. Similarly, **ChooseAction** returns to the method selection point and selects the next method, when the set of actions becomes empty during the matchings or by the evaluation of the application conditions. If **ChooseAction** fails to compute an action that does not correspond to a backtracked action and is not rejected by the control rules, then it terminates and returns **fail** (see step 2 in Figure 4.11).

Chapter 5

A Short Introduction to the Case Studies

In this chapter we shall introduce the *limit domain* [169, 168, 172] and the *residue class domain* [166, 163, 165] for which we conducted in-depth case studies for the application of Multi. The limit domain was first tackled with the previous planner PLAN whose application was successful for many theorems but failed on some typical ones. The analysis of the failed attempts of PLAN strongly influenced the design of Multi. The residue class domain was directly tackled with Multi.

Detailed discussions on how Multi tackles problems of these domains can be found in chapter 8 and chapter 9. We briefly introduce both domains already here since we shall use examples from both domains to motivate and discuss the Multi system throughout the remainder of the thesis.

5.1 The Limit Domain

In the following, we shall explain proof planning *limit theorems* and their relatives. These theorems are formulated and proved in the theory $\mathbb{R}$ of the real numbers. In the remainder of this thesis, $/_{\nu\nu\nu}, *_{\nu\nu\nu}, +_{\nu\nu\nu}, -_{\nu\nu\nu}, ||_{\nu\nu}$ denote the division, multiplication, addition, subtraction, and the absolute value function in $\mathbb{R}$, respectively.

Theorems of the *limit domain* make statements about the limit $\lim\limits_{x\to a} f(x)$ of a function f at a point a, about the limit $limseq\, X$ of a sequence X, about the continuity of a function f at a point a, and about the derivative of a function f at a point a. Since the standard definitions of limit, continuity, and derivative are

$$\begin{array}{lcl}
lim_{(\nu\nu)\nu\nu o} & \equiv & \begin{array}{l}\lambda f_{\nu\nu}. \lambda a_{\nu}. \lambda l_{\nu}. \forall \epsilon_{\nu}. (0 < \epsilon \Rightarrow \\ \qquad \exists \delta_{\nu}. (0 < \delta \wedge \forall x_{\nu}. (|x-a| > 0 \wedge |x-a| < \delta \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad |f(x) - l| < \epsilon)))\end{array} \\
limseq_{(\nu\nu)\nu o} & \equiv & \begin{array}{l}\lambda X_{\nu\nu}. \lambda l_{\nu}. \forall \epsilon_{\nu}. (0 < \epsilon \Rightarrow \\ \quad \exists k_{\nu}. (k \in \mathbb{N} \wedge \forall n_{\nu}. (n \in \mathbb{N} \wedge n > k \Rightarrow |(X\ n) - l| < \epsilon)))\end{array} \\
cont_{(\nu\nu)\nu o} & \equiv & \begin{array}{l}\lambda f_{\nu\nu}. \lambda a_{\nu}. \forall \epsilon_{\nu}. (0 < \epsilon \Rightarrow \\ \quad \exists \delta_{\nu}. (0 < \delta \wedge \forall x_{\nu}. (|x-a| < \delta \Rightarrow |f(x) - f(a)| < \epsilon)))\end{array} \\
deriv_{(\nu\nu)\nu\nu o} & \equiv & \lambda f_{\nu\nu}. \lambda a_{\nu}. \lambda f'_{\nu}. \lim\limits_{x \to a} \frac{f(x)-f(c)}{x-c}
\end{array}$$

the proofs of these theorems are so-called *ϵ-δ-proofs*, i.e., proofs that postulate the existence of a δ such that a conjecture of the form $\ldots |X| < \epsilon$ is proved under assumptions of the form $\ldots |Y| < \delta$.

Notation 5.1: Instead of the formula $lim(f_{\nu\nu}, a_\nu, l_\nu)$ we henceforth write the more common equation expression $\lim_{x\to a} f(x) = l$. Analogously, we write $limseq X = l$ instead of $limseq(X_{\nu\nu}, l_\nu)$ and $deriv(f, a) = f'$ instead of $deriv(f_{\nu\nu}, a_\nu, f'_\nu)$.

An example theorem from the limit domain is LIM+ that states that the limit of the sum of two functions f and g equals the sum of their limits; that is, if $\lim_{x\to a} f(x) = l_1$ and $\lim_{x\to a} g(x) = l_2$ then $\lim_{x\to a}(f(x) + g(x)) = l_1 + l_2$. When the definition of $\lim_{x\to a}$ is expanded, the corresponding planning problem consists of two assumptions

$$\forall \epsilon_1 . (0 < \epsilon_1 \Rightarrow \exists \delta_1 . (0 < \delta_1 \wedge \forall x_1 . (|x_1 - a| > 0 \wedge |x_1 - a| < \delta_1 \Rightarrow |f(x_1) - l_1| < \epsilon_1)))$$

and

$$\forall \epsilon_2 . (0 < \epsilon_2 \Rightarrow \exists \delta_2 . (0 < \delta_2 \wedge \forall x_2 . (|x_2 - a| > 0 \wedge |x_2 - a| < \delta_2 \Rightarrow |g(x_2) - l_2| < \epsilon_2))).$$

And the theorem becomes

$$\forall \epsilon . (0 < \epsilon \Rightarrow \exists \delta . (0 < \delta \wedge \forall x . (|x - a| > 0 \wedge |x - a| < \delta \\ \Rightarrow |(f(x) + g(x)) - (l_1 + l_2)| < \epsilon))).$$

Similar theorems in this class are LIM- and LIM* for the difference and the product of limits of functions. Moreover, there are corresponding theorems about continuity. Continuous+ states that the sum of two continuous functions is continuous, and Continuous- and Continuous* make similar statements for the difference and product of continuous functions. We shall introduce some further examples from the limit domain in the remainder of the thesis.

When proving a limit theorem like LIM+, a δ has to be constructed that depends on an ϵ such that certain estimations hold. This is a non-trivial task for students as well as for traditional automated theorem provers.[1] The typical way a mathematician discovers a suitable δ is by incrementally restricting the possible values of δ. When proof planning limit theorems, PLAN adapts this approach by cooperating with the constraint solver $\mathcal{C}o\mathcal{SIE}$: (in)equality tasks that are simple enough for $\mathcal{C}o\mathcal{SIE}$ (i.e., tasks that are in the input language for $\mathcal{C}o\mathcal{SIE}$) are passed to $\mathcal{C}o\mathcal{SIE}$ and $\mathcal{C}o\mathcal{SIE}$ provides suitable instantiations for δ, when solutions for meta-variables are computed and inserted into the final proof plan.

For finding ϵ-δ-proofs, among others, the general methods ∃I-B, ∃E-F, ∀I-B, ∀E-F, ∧I-B, ∧E-F, ⇒I-B, ⇒E-F, and =Subst-B and the domain-specific methods TellCS-B, TellCS-F, AskCS-B, Solve*-B, and ComplexEstimate-B are required. We introduced ∀I-B, ∃E-F, AskCS-B, TellCS-B, TellCS-F, and ComplexEstimate-B already in section 4.1.4; =Subst-B is explained already in section 4.1.1. Similar to ∀I-B and ∃E-F also ∃I-B, ∀E-F, ∧I-B, ∧E-F, ⇒I-B, and ⇒E-F apply certain natural deduction rules. Actions of ∃I-B perform a backward $\exists_I$ step. They close a goal with formula $\exists x . P[x]$ and introduce a task whose goal has the formula $P[mv]$ in which x is replaced by a new meta-variable mv. Similarly, actions of ∀E-F perform a forward $\forall_E$ step and derive a new support $P[mv]$ with a new meta-variable mv from a given support $\forall x . P[x]$. Actions of ∧I-B perform a backward $\wedge_I$ step and reduce a task whose goal has the formula $A_1 \wedge A_2$ to new tasks whose goals have the formulas A_1 and A_2. Actions of ∧E-F perform the corresponding forward $\wedge_E$ decompositions on conjunctive support lines. Actions of ⇒I-B perform a backward $\Rightarrow_I$ step and reduce a task with goal $A \Rightarrow B$ to a new task whose goal has the formula B and A as additional hypothesis. Moreover, A becomes the formula of a new support for this task. Actions of ⇒E-F perform

[1] Bledsoe proposed in 1990 several versions of LIM+ as a challenge problem for automated theorem proving [28]. The simplest versions of LIM+ (problem 1 and 2 in [28]) are at the edge of the capabilities of traditional automated theorem provers but LIM* is certainly beyond their capabilities.

an $\Rightarrow_E$ step. When applied to a task with goal C and an support with formula $A \Rightarrow B$ they introduce two new tasks: a task with goal C, which contains also a new support with B as formula, and a task with goal A. Actions of the SOLVE*-B method exploit transitivity of $<, >, \leq, \geq$ and reduce a goal with formula $a_1 < b_1$ to a new task with formula $b_2\sigma \leq b_1\sigma$ in case a support $a_2 < b_2$ exists and a_1, a_2 can be unified by the substitution σ. Then, also a further new task is created whose formula is the conjunction of all mappings of the substitution σ (compare description of method COMPLEXESTIMATE-B in section 4.1.4).

When applied to an ϵ–δ–problem, PLAN first decomposes the initial task with a complex formula into subtasks whose formulas are (in)equalities. This is done by actions that decompose formulas in tasks, e.g., actions of the methods $\wedge$I-B, $\forall$I-B, $\exists$I-B etc. Then, tasks whose formulas are simple (in)equalities are closed by actions of TELLCS-B and their formulas are passed as constraints to $\mathcal{C}o\mathcal{SIE}$. Tasks, which follow from the constraints collected by $\mathcal{C}o\mathcal{SIE}$, are closed by actions of ASKCS-B. In order to satisfy more complex tasks, the unwrapping of (in)equality supports from the initial assumptions is necessary. This is realized by actions that decompose supports, e.g., actions of the methods $\wedge$E-F, $\forall$E-F, $\exists$E-F etc. The introduction of these actions results in (in)equality supports that can be used to further tackle tasks with complex formulas with actions of the methods SOLVE*-B or COMPLEXESTIMATE-B. By actions of these methods tasks whose formulas are complex (in)equalities are successively reduced to tasks whose formulas are simple (in)equalities that can be closed and passed to $\mathcal{C}o\mathcal{SIE}$ by actions of TELLCS-B. Finally, when no task is left and PLAN invokes the function *employ-CS*, $\mathcal{C}o\mathcal{SIE}$ computes instantiations for the meta-variables that are consistent with the collected constraints.

Next, we briefly discuss the application of PLAN to the LIM+ problem.[2] PLAN first decomposes the initial theorem to tasks with the formulas $0 < mv_\delta$ and $|(f(c_x)+g(c_x))-(l_1+l_2)| < c_\epsilon$ where mv_δ is a meta-variable introduced for δ and c_x and c_ϵ are constants that replace x and ϵ, respectively. Moreover, the assumptions $0 < c_\epsilon$, $|c_x - a| > 0$, and $|c_x - a| < mv_\delta$ are created during the decomposition of the initial theorem and become supports of the new tasks. $0 < mv_\delta$ can be passed directly to $\mathcal{C}o\mathcal{SIE}$ by an action of TELLCS-B. $|(f(c_x)+g(c_x))-(l_1+l_2)| < c_\epsilon$ cannot be passed to $\mathcal{C}o\mathcal{SIE}$ directly. This triggers the decomposition of one of the two initial assumptions. If the initial assumption on f is decomposed, then PLAN obtains as new supports $0 < c_{\delta_1}$ and $|f(mv_{x_1}) - l_1| < mv_{\epsilon_1}$. Now PLAN can compute and introduce an action of COMPLEXESTIMATE-B using the latter new support line. During the evaluation of the application conditions of COMPLEXESTIMATE-B the substitution $mv_{x_1} \mapsto c_x$ is created and the computer algebra system MAPLE computes a decomposition $(f(c_x)+g(c_x))-(l_1+l_2) = 1*(f(c_x)-l_1)+(g(c_x)+l_2)$ (that is, the variables k and l of COMPLEXESTIMATE-B are bound to 1 and $g(c_x) - l_2$, respectively). Thus, the action of COMPLEXESTIMATE-B introduces new tasks with formulas $mv_{\epsilon_1} < \frac{c_\epsilon}{2*mv}$, $|1| \leq mv$, $0 < mv$, $|g(c_x) - l_2| < \frac{c_\epsilon}{2}$, and $mv_{x_1} \doteq c_x$. The formulas of the former three tasks and of the last one can all be passed directly to $\mathcal{C}o\mathcal{SIE}$ by actions of TELLCS-B. To deal with the remaining task with formula $|g(c_x) - l_2| < \frac{c_\epsilon}{2}$ PLAN decomposes the second initial assumption (on g) and derives new support lines with formulas $0 < c_{\delta_2}$ and $|g(mv_{x_2}) - l_2| < mv_{\epsilon_2}$. An action of SOLVE*-B reduces the goal with respect to the second new support to two new tasks with formulas $mv_{\epsilon_2} \leq \frac{c_\epsilon}{2}$ and $mv_{x_2} \doteq c_x$. Both tasks are closed by actions of TELLCS-B and their formulas are passed to $\mathcal{C}o\mathcal{SIE}$.

The decomposition of the initial assumptions results not only in the used support lines but also in tasks with the formulas $0 < mv_{\epsilon_1}$, $|mv_{x_1} - a| > 0$, $|mv_{x_1} - a| < c_{\delta_1}$

[2] A detailed description on how MULTI solves this problem is given in section 8.1.

from the assumption on f and the analogue tasks from the assumption on g. The task $0 < mv_{\epsilon_1}$ is closed by the introduction of an action of TELLCS-B, which passes the formula to $\mathcal{C}o\mathcal{SIE}$. To close the other tasks PLAN introduces actions of the method SOLVE*-B that use the supports with formulas $|c_x - a| < mv_\delta$ and $|c_x - a| > 0$ (from the decomposition of the initial goal). The application of SOLVE*-B to the task $|mv_{x_1} - a| < c_{\delta_1}$ and the support $|c_x - a| < mv_\delta$ results in two new tasks with formulas $mv_\delta \leq c_{\delta_1}$ and $mv_{x_1} \doteq c_x$. The application of SOLVE*-B to the task $|mv_{x_1} - a| > 0$ and the support $|c_x - a| > 0$ results also in two new tasks with formulas $0 \leq 0$ and $mv_{x_1} \doteq c_x$. Whereas $0 \leq 0$ is closed by an actions of ASKCS-B the other three tasks are closed by actions of TELLCS-B, which pass their formulas to $\mathcal{C}o\mathcal{SIE}$. The corresponding tasks from the assumption on g are handled in the same way. Thereby the constraints $mv_\delta \leq c_{\delta_2}$, $mv_{x_2} \doteq c_x$, and $mv_{x_2} \doteq c_x$ are passed to $\mathcal{C}o\mathcal{SIE}$. Moreover, some actions of the TELLCS-F method during the planning process pass constraints in support lines to $\mathcal{C}o\mathcal{SIE}$: $0 < c_{\delta_1}$, $0 < c_{\delta_2}$, $0 < c_\epsilon$.

After propagating constraints, $\mathcal{C}o\mathcal{SIE}$ has the final constraint store in Figure 5.1. When asked for suitable instantiations for the meta-variables, $\mathcal{C}o\mathcal{SIE}$ provides the bindings $mv_{x_1} \mapsto c_x, mv_{x_2} \mapsto c_x, mv \mapsto 1$, $mv_{\epsilon_1} \mapsto \frac{c_\epsilon}{2}$, $mv_{\epsilon_2} \mapsto \frac{c_\epsilon}{2}$, and $mv_\delta \mapsto min(c_{\delta_1}, c_{\delta_2})$. These instantiations computed by $\mathcal{C}o\mathcal{SIE}$ are exactly the solutions that standard textbooks use for δ, ϵ_1, and ϵ_2 for LIM+.

$$
\begin{array}{lllll}
mv_{x_1} = c_x & & & & \\
mv_{x_2} = c_x & & & & \\
0 & < & c_{\delta_1} & < & +\infty \\
0 & < & c_{\delta_2} & < & +\infty \\
0 & < & c_\epsilon & < & +\infty \\
0 & < & mv_{\epsilon_1} & \leq & \frac{c_\epsilon}{2}, \frac{c_\epsilon}{2*mv} \\
0 & < & mv_{\epsilon_2} & \leq & \frac{c_\epsilon}{2} \\
0 & < & mv_\delta & \leq & c_{\delta_1}, c_{\delta_2} \\
1 & \leq & mv & \leq & \frac{c_\epsilon}{2*mv_{\epsilon_1}}
\end{array}
$$

Figure 5.1: The final constraint store of $\mathcal{C}o\mathcal{SIE}$ for LIM+.

PLAN can successfully plan all the challenge problems of BLEDSOE [28], i.e., the limit theorems LIM+, LIM-, LIM*, the theorems Continuous+, Continuous−, Continuous*, $\lim_{x \to a} x = a$, $\lim_{x \to a} c = c$, and the theorem that the composition of continuous functions is again continuous. Moreover, we tried to apply PLAN to tackle systematically the limit problems recorded in the textbook of BARTLE and SHERBERT "Introduction to Real Analysis" [12]. A summary of these experiments can be found in the master thesis of Jürgen Zimmer [255]. It turned out that PLAN failed to plan several theorems from [12]. As we shall discuss in the next chapter when motivating the development of MULTI this is not due to missing or inappropriate methods but due to PLAN's inadequate algorithm.

5.2 The Residue Class Domain

In the following, we shall introduce the *residue class domain*. The theorems of this domain are formulated and proved in the theories $\mathbb{Z}$ of integers and *Group*. Since this case study was directly performed with MULTI we give no description on how PLAN tackles residue class problems.

5.2.1 An Informal Introduction to the Residue Class Domain

In the residue class domain we are interested in proving properties of mathematical structures consisting of residue class sets over the integers and binary operations. First we examine their basic algebraic properties to classify the structures in terms of group, monoid etc. Moreover, we are interested in proving that two structures are isomorphic or not.

A *residue class set* over the integers is either the set of all congruence classes modulo 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 $cl_n(c)$ 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-notation. $\circ$ can be of the form $\lambda xy{\bullet}x$, $\lambda xy{\bullet}y$, $\lambda xy{\bullet}c$, where c is a constant congruence class (e.g., $\bar{1}_3$), $\lambda xy{\bullet}x\bar{+}y$, $\lambda xy{\bullet}x\bar{*}y$, $\lambda xy{\bullet}x\bar{-}y$, 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, $\lambda xy{\bullet}(x\bar{+}\bar{1}_3)\bar{-}(y\bar{+}\bar{2}_3)$. For direct products of residue class sets the operation is a combination of the single binary operations for the element tuples, for example, $\lambda xy{\bullet}x\bar{+}y \oplus \lambda xy{\bullet}x\bar{*}y$.

We are interested in algebraic properties of residue class sets RS_n modulo n either with one binary operation $\circ$ or with 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. Such a mathematical structure consisting of a residue class set with one or two binary operations is called a *residue class structure* (or simply *structure*) and is denoted by $(RS_n, \circ)$ or $(RS_n, \circ, \star)$, respectively. For structures with one operation, $(RS_n, \circ)$, we are interested in classifying them in terms of magma, semi-group, monoid, quasi-group, loop, or group and whether they are Abelian. To determine the algebra of a structure we have to prove or to refute some of the following properties:

1. **Closure:** RS_n is closed under $\circ$. This is formalized by the defined concept $closed(RS_n, \circ)$ that abbreviates $\forall x{:}_{RS_n}{\bullet}\forall y{:}_{RS_n}{\bullet}(x \circ y) \in RS_n$.
2. **Associativity:** RS_n is associative with respect to $\circ$.
   ($assoc(RS_n, \circ) \equiv \forall x{:}_{RS_n}{\bullet}\forall y{:}_{RS_n}{\bullet}\forall z{:}_{RS_n}{\bullet}x \circ (y \circ z) \doteq (x \circ y) \circ z$.)
3. **Unit element:** There exists a unit element in RS_n with respect to $\circ$.
   ($\exists e{:}_{RS_n}{\bullet}unit(RS_n, \circ, e) \equiv \exists e{:}_{RS_n}{\bullet}\forall y{:}_{RS_n}{\bullet}[y \circ e \doteq y] \wedge [e \circ y \doteq y]$.)
4. **Inverses:** Every element in RS_n has an inverse element with respect to $\circ$ and the unit element e.
   ($inverse(RS_n, \circ, e) \equiv \forall x{:}_{RS_n}{\bullet}\exists y{:}_{RS_n}{\bullet}[x \circ y \doteq e] \wedge [y \circ x \doteq e]$.)
5. **Divisors:** For every two elements of RS_n there exist two corresponding divisors in RS_n with respect to $\circ$.
   ($divisors(RS_n, \circ) \equiv \forall a{:}_{RS_n}{\bullet}\forall b{:}_{RS_n}{\bullet}[\exists x{:}_{RS_n}{\bullet}a \circ x \doteq b] \wedge [\exists y{:}_{RS_n}{\bullet}y \circ a \doteq b]$.)
6. **Commutativity:** RS_n is commutative with respect to $\circ$.
   ($commu(RS_n, \circ) \equiv \forall a{:}_{RS_n}{\bullet}\forall b{:}_{RS_n}{\bullet}a \circ b \doteq b \circ a$.)

Given a structure $(RS_n, \circ, \star)$ consisting of a residue class set and two binary operations, first we can determine its category with respect to each operation separately. Then, we check the property of distributivity

7. **Distributivity:** RS_n is distributive with respect to $\circ$ and $\star$.
$(distrib(RS_n,\circ,\star)\equiv\forall a{:}_{RS_n}.\forall b{:}_{RS_n}.\forall c{:}_{RS_n}.[a\star(b\circ c)\doteq(a\star b)\circ(a\star c)]\wedge[(a\circ b)\star c\doteq(a\star c)\circ(b\star c)].)$

We can classify $(RS_n,\circ,\star)$ in terms of ring, ring-with-one, division ring, or field. The proof problems resulting from the properties 1 to 7 are called the *simple residue class problems*.

Furthermore, we are interested in distinguishing classes of isomorphic structures. Two structures $(RS^1_{n_1},\circ_1)$ and $(RS^2_{n_2},\circ_2)$ are isomorphic, if and only if there exists a function $h:RS^1_{n_1}\rightarrow RS^2_{n_2}$, such that h is an injective and surjective homomorphism. Thus, we have to prove or to refute the following property:

8. **Isomorphic:** $(RS^1_{n_1},\circ_1)$ and $(RS^2_{n_2},\circ_2)$ are isomorphic.
$(Iso(RS^1_{n_1},\circ_1,RS^2_{n_2},\circ_2)\equiv\exists h{:}_{F(RS^1_{n_1},RS^2_{n_2})}.Inj(h,RS^1_{n_1})\wedge$
$Surj(h,RS^1_{n_1},RS^2_{n_2})\wedge Hom(h,RS^1_{n_1},\circ_1,RS^2_{n_2},\circ_2)$,
where $F(RS^1_{n_1},RS^2_{n_2})$ is the set of all total functions from $RS^1_{n_1}$ into $RS^2_{n_2}$,
$Inj(h,RS^1_{n_1})\equiv\forall x_1{:}_{RS^1_{n_1}}.\forall x_2{:}_{RS^1_{n_1}}.h(x_1)\doteq h(x_2)\Rightarrow x_1\doteq x_2$,
$Surj(h,RS^1_{n_1},RS^2_{n_2})\equiv\forall x{:}_{RS^2_{n_2}}.\exists y{:}_{RS^1_{n_1}}.h(y)\doteq x$,
$Hom(h,RS^1_{n_1},\circ_1,RS^2_{n_2},\circ_2)\equiv$
$\forall x_1{:}_{RS^1_{n_1}}.\forall x_2{:}_{RS^1_{n_1}}.h(x_1\circ_1 x_2)\doteq h(x_1)\circ_2 h(x_2).)$

5.2.2 Formalizations of Concepts in the Residue Class Domain

First, we formalize in λ-calculus the simple properties used in the preceding section.

$$Closed_{(\beta o)(\beta\beta\beta)o} \equiv \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\forall a_\beta{:}_G.\forall b_\beta{:}_G.G(a\circ b) \tag{5.1}$$

$$\begin{aligned} Assoc_{(\beta o)(\beta\beta\beta)o} \equiv{}& \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\forall a_\beta{:}_G.\forall b_\beta{:}_G.\forall c_\beta{:}_G.\\ &(a\circ(b\circ c))\doteq((a\circ b)\circ c) \end{aligned} \tag{5.2}$$

$$\begin{aligned} Unit_{(\beta o)(\beta\beta\beta)o} \equiv{}& \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\lambda e_\beta.\\ &\forall a_\beta{:}_G.[(a\circ e)\doteq a]\wedge[(e\circ a)\doteq a] \end{aligned} \tag{5.3}$$

$$\begin{aligned} Inverse_{(\beta o)(\beta\beta\beta)o} \equiv{}& \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\lambda e_\beta.\\ &\forall a_\beta{:}_G.\exists x_\beta{:}_G.[(a\circ x)\doteq e]\wedge[(x\circ a)\doteq e] \end{aligned} \tag{5.4}$$

$$\begin{aligned} Divisors_{(\beta o)(\beta\beta\beta)o} \equiv{}& \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\forall a_\beta{:}_G.\forall b_\beta{:}_G.\\ &[\exists x_\beta{:}_G.(a\circ x)\doteq b]\wedge[\exists y_\beta{:}_G.(y\circ a)\doteq b] \end{aligned} \tag{5.5}$$

$$Commu_{(\beta o)(\beta\beta\beta)o} \equiv \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\forall a_\beta{:}_G.\forall b_\beta{:}_G.[(a\circ b)\doteq(b\circ a)] \tag{5.6}$$

$$\begin{aligned} Distrib_{(\beta o)(\beta\beta\beta)(\beta\beta\beta)o} \equiv{}& \lambda G_{\beta o}.\lambda\circ_{\beta\beta\beta}.\lambda\star_{\beta\beta\beta}.\forall a_\beta{:}_G.\forall b_\beta{:}_G.\forall c_\beta{:}_G.\\ &[(a\star(b\circ c))\doteq((a\star b)\circ(a\star c))]\\ &\wedge[((a\circ b)\star c)\doteq((a\circ c)\star(b\circ c))] \end{aligned} \tag{5.7}$$

The concepts for isomorphism problems are formalized as follows.

$$\begin{aligned} Hom_{(\alpha\beta)(\alpha o)(\alpha\alpha\alpha)(\beta o)(\beta\beta\beta)o} \equiv{}& \lambda h_{\alpha\beta}.\lambda A_{\alpha o}.\lambda\circ_{\alpha\alpha\alpha}.\lambda B_{\beta o}.\lambda\star_{\beta\beta\beta}.\\ &\forall x_\alpha{:}_A.\forall y_\alpha{:}_A.h(x\circ y)\doteq h(x)\star h(y) \end{aligned} \tag{5.8}$$

$$Inj_{(\alpha\beta)(\alpha o)o} \equiv \lambda f_{\alpha\beta}.\lambda A_{\alpha o}. \tag{5.9}$$

$$\forall x_\alpha{:}_A.\forall y_\alpha{:}_A.f(x)\doteq f(y)\Rightarrow x\doteq y \tag{5.10}$$

$$Surj_{(\alpha\beta)(\alpha o)(\beta o)o} \equiv \lambda f_{\alpha\beta}.\lambda A_{\alpha o}.\lambda B_{\beta o}.\forall x_\beta{:}_B.\exists y_\alpha{:}_A.f(y)\doteq x \tag{5.11}$$

$$Iso_{(\alpha o)(\alpha\alpha\alpha)(\beta o)(\beta\beta\beta)o} \equiv \lambda A_{\alpha o}.\lambda\circ_{\alpha\alpha\alpha}.\lambda B_{\beta o}.\lambda\star_{\beta\beta\beta}.\exists h{:}_{F(A,B)}. \tag{5.12}$$

$$\begin{aligned} &Inj(h,A)\wedge Surj(h,A,B)\wedge\\ &Hom(h,A,\circ,B,\star) \end{aligned} \tag{5.13}$$

Next, we formalize the notion of a *congruence class* and a *residue class set*. We start with the basic notion of a congruence class:

$$cl_{\nu\nu\nu o} \equiv \lambda n_{\nu}.\lambda m_{\nu}.\lambda x_{\nu}.[\mathbb{Z}(x)] \wedge [(x \; mod \; n) \doteq m] \tag{5.14}$$

Provided, *cl* is applied to two arguments n and m, the resulting set contains all integers x such that $(x \; mod \; n) \doteq m$. 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 meaningful only, if n is an integer, which in particular is not zero. Having congruence classes as building blocks available we can define residue class sets as

$$RS_{\nu(\nu o)o} \equiv \lambda n_{\nu}.\lambda r_{\nu o}.\exists m_{\nu}{:}\mathbb{N}.[r \doteq cl_n(m)] \wedge [NonEmpty(cl_n(m))]. \tag{5.15}$$

A residue class set over an integer n, that is, the application of RS to an integer n, is denoted by RS_n (as introduced in the preceding section).

The basic operations $\bar{+}, \bar{*}, \bar{-}$ on congruence classes are defined as follows:

$$\bar{+}_{(\nu o)(\nu o)\nu o} \equiv \lambda r_{\nu o}.\lambda s_{\nu o}.\lambda z_{\nu}.\exists x_{\nu}{:}r.\exists y_{\nu}{:}s.z \doteq x + y \tag{5.16}$$

$$\bar{*}_{(\nu o)(\nu o)\nu o} \equiv \lambda r_{\nu o}.\lambda s_{\nu o}.\lambda z_{\nu}.\exists x_{\nu}{:}r.\exists y_{\nu}{:}s.z \doteq x * y \tag{5.17}$$

$$\bar{-}_{(\nu o)(\nu o)\nu o} \equiv \lambda r_{\nu o}.\lambda s_{\nu o}.\lambda z_{\nu}.\exists x_{\nu}{:}r.\exists y_{\nu}{:}s.z \doteq x - y \tag{5.18}$$

These definitions (5.16) – (5.18) make no restriction on the congruence classes. For instance, they do not have to be congruence classes with respect to the same modulo factor. However, in practice, operations between congruence classes with differing modulo factor are meaningless.

With respect to definition 5.15 the type of a residue class set RS_n is $(\nu o)o$. Moreover, with respect to the definitions 5.16 – 5.18 the type of a binary operation on residue classes is $(\nu o)(\nu o)\nu o$ (i.e., the basic binary operations on residue classes given in the preceding section are: $\lambda x_{\nu o} y_{\nu o}.x$, $\lambda x_{\nu o} y_{\nu o}.y$, $\lambda x_{\nu o} y_{\nu o}.c_{\nu o}$, $\lambda x_{\nu o} y_{\nu o}.x \bar{+} y$, $\lambda x_{\nu o} y_{\nu o}.x \bar{*} y$, $\lambda x_{\nu o} y_{\nu o}.x \bar{-} y$ when completely typed). The definitions 5.1 – 5.7 specify the concepts *closed*, *assoc* etc. for a general set $G_{\beta o}$ and a general binary operation $\circ_{\beta\beta\beta}$ on G using the type-variable β. When applied to a residue class set RS_n and a binary operation on residue classes, β is instantiated by νo. Similarly, the α and the β in the definitions 5.8 – 5.13 are instantiated by νo, when the corresponding concepts are applied to residue class sets and operations.

Part II

MULTI

Chapter 6

Basics of Proof Planning with Multiple Strategies

The development of proof planning with multiple strategies was motivated by problems we encountered with the PLAN proof planner, when we extended the exploration of the limit domain and when we explored further domains. These problems caused a reconsideration of ΩMEGA's proof planning approach and gave rise to the development of multi-strategy proof planning, which we realized in the MULTI system.

In this chapter, we shall introduce the basic notions of proof planning with multiple strategies and discuss the blackboard architecture of the MULTI system. This blackboard architecture reflects a paradigm shift for proof planning from the rigid precondition achievement paradigm on which PLAN is based to a problem solving process in which independent components for different kinds of plan refinements and modifications can flexibly cooperate guided by meta-reasoning on their applicability and desirability in a given situation.

The structure of the chapter is as follows. As motivation we start with a discussion of the drawbacks of PLAN and compare our observations with mathematical experience. In section 6.2, we introduce the basic concepts of proof planning with multiple strategies and describe MULTI's blackboard architecture. Afterwards, we discuss the design decisions of MULTI and compare the MULTI approach with related work.

6.1 Motivation

The problem solving approach of the previous planner PLAN is hard-coded into its algorithm in several aspects. First, the three components for action introduction, backtracking, and meta-variable instantiation are employed in a pre-defined way: As long as there are tasks, PLAN tries to introduce actions to tackle the tasks; it performs backtracking if and only if it encounters a task for which it fails to compute an action; it delays the instantiation of meta-variables until all planning tasks are solved. Second, the capabilities of the single components of PLAN are limited: The backtracking component performs only dependency directed backtracking that removes the action that introduced the task for which no action was found. The component for meta-variable instantiation employs only external constraints solvers to compute instantiations for meta-variables based on collected constraints. Finally, the component for action introduction always performs the same cycle of action computation and selection.

In the following, we shall discuss some examples and scenarios that show the drawbacks of this hard-coded problem solving approach. Together with the drawbacks, we shall also analyze what functionalities are necessary to overcome the problems. In particular, we shall discuss available domain knowledge that could be useful but cannot be employed by PLAN since it is beyond the means of methods and control rules. Finally, we shall compare our observations with mathematical experience.

6.1.1 Flexible Meta-Variable Instantiation

PLAN instantiates meta-variables only if all tasks are closed. Moreover, it employs only constraints solvers to obtain instantiations for meta-variables. These restrictions cause that PLAN fails on some problems since it cannot make use of available knowledge of suitable instantiations to simplify the problems.

For instance, consider exercise 4.1.3 in the analysis textbook [12].

Exercise 4.1.3 Let $f : \mathbb{R} \to \mathbb{R}$ and let $c \in \mathbb{R}$. Show that $\lim_{x_1 \to c} f(x_1) = l$ if and only if $\lim_{x \to 0} f(x + c) = l$.

Two implications have to be proof planned for solving this exercise:

$$\lim_{x_1 \to c} f(x_1) = l \quad \Rightarrow \quad \lim_{x \to 0} f(x + c) = l \tag{6.1}$$

and

$$\lim_{x \to 0} f(x + c) = l \quad \Rightarrow \quad \lim_{x_1 \to c} f(x_1) = l \tag{6.2}$$

With respect to the definition of limit given in section 5.1 for (6.1) we need to show that

$$\forall \epsilon \centerdot (0 < \epsilon \Rightarrow \exists \delta \centerdot (0 < \delta \wedge \forall x \centerdot (|x - 0| > 0 \wedge |x - 0| < \delta \Rightarrow |f(x + c) - l| < \epsilon)))$$

holds under the assumption that

$$\forall \epsilon_1 \centerdot (0 < \epsilon_1 \Rightarrow \exists \delta_1 \centerdot (0 < \delta_1 \wedge \forall x_1 \centerdot (|x_1 - c| > 0 \wedge |x_1 - c| < \delta_1 \Rightarrow |f(x_1) - l| < \epsilon_1))).$$

PLAN first decomposes the task formula. This results in new tasks with formulas $0 < mv_\delta$ and $|f(c_x + c) - l| < c_\epsilon$ and new supports with formulas $|c_x - 0| < mv_\delta$ and $|c_x - 0| > 0$ where mv_δ is a meta-variable and c_x and c_ϵ are constants. The new task with formula $0 < mv_\delta$ can be directly closed with an action of TELLCS-B. The formula $|f(c_x + c) - l| < c_\epsilon$ of the other task is too complex to be sent to $\mathcal{C}o\mathcal{SIE}$ directly. Hence PLAN unwraps the assumption which results in a new support with formula $|f(mv_{x_1}) - l| < mv_{\epsilon_1}$ as well as two new tasks with formulas $|mv_{x_1} - c| < c_\delta$ and $|mv_{x_1} - c| > 0$. Now the task with formula $|f(c_x + c) - l| < c_\epsilon$ can be closed by an action of SOLVE*-B that uses the new support. This action yields new tasks with the formulas $mv_{\epsilon_1} \leq c_\epsilon$ and $mv_{x_1} \doteq c_x + c$, which both can be closed and passed to $\mathcal{C}o\mathcal{SIE}$ by actions of TELLCS-B.

The tasks with formulas $|mv_{x_1} - c| < c_{\delta_1}$ and $|mv_{x_1} - c| > 0$ should be closed by the method SOLVE*-B using the supports $|c_x - 0| < mv_\delta$ and $|c_x - 0| > 0$. However, SOLVE*-B is not applicable and hence proof planning is blocked because $(mv_{x_1} - c)$ and $(c_x - 0)$ cannot be unified. If PLAN could use the information that $c_x + c$

is the (only) suitable instantiation for mv_{x_1} available in the constraint store, then an eager instantiation of mv_{x_1} by $c_x + c$ would unblock the planning because the formulas of the task would be instantiated to $|c_x + c - c| < c_{\delta_1}$ and $|c_x + c - c| > 0$. Then, the tasks could be reduced to tasks with the simplified formulas $|c_x| < c_{\delta_1}$ and $|c_x| > 0$ to which SOLVE*-B would be applicable using the simplified supports $|c_x| < mv_\delta$ and $|c_x| > 0$ that are implied by $|c_x - 0| < mv_\delta$ and $|c_x - 0| > 0$.[1]

The lack of the flexibility to instantiate meta-variables during the planning process whenever needed or beneficial (even if there are still tasks) is one problem of PLAN. The other problem is that the computation of instantiations is restricted to constraint solvers (i.e., to $\mathcal{C}o\mathcal{SIE}$). In other domains, however, there can be other means providing suitable instantiations for meta-variables. For instance, consider the problems of the residue class domain: many of these problems postulate the existence of elements of the involved residue class sets that have some special properties. For instance, when classifying the structure $(\mathbb{Z}_5, \bar{+})$ as a monoid we have to prove — among other things — that the structure has a unit element: $\exists e{:}\mathbb{Z}_5 \centerdot \forall y{:}\mathbb{Z}_5 \centerdot [y \bar{+} e = y] \wedge [e \bar{+} y = y]$. In the planning process the existentially quantified variable is substituted by a meta-variable. Proof planning such problems becomes considerably easier, if suitable instantiations for the meta-variables can be provided early in the proof by external oracles. In the residue class domain, computer algebra systems turned out to be our main knowledge source for instantiations rather than constraint solvers. When proof planning for the problem given above, a meta-variable mv_e is introduced for e. When we pass the structure $(\mathbb{Z}_5, \bar{+})$ to the computer algebra system GAP [93], a system specialized on algebra, then GAP can directly provide the solution $\bar{0}_5$. The instantiation of mv_e by $\bar{0}_5$ reduces the problem at hand to the problem to show that this is the right instantiation instead of showing that there is a suitable instantiation at all.

The lesson learned from these and similar examples is that we need heterogeneous knowledge sources for the computation of substitutes for meta-variables. Moreover, these knowledge sources should be flexibly employed whenever needed or beneficial during the proof planning process rather than at the end only.

6.1.2 Flexible Backtracking and Reasoning on Failures

If a task occurs for which PLAN fails to compute an applicable action (we call this situation a *failure*), then PLAN's only remedy is dependency directed backtracking by deleting the action that introduced this task. Moreover, failures are the only events that trigger backtracking in PLAN. These restrictions cause that PLAN fails on some problems and that it cannot make use of knowledge of how to deal and productively make use of failures.

For instance, consider knowledge of where to backtrack. Suppose an action A is introduced during the planning process, which leads into a search branch that ends with a task T for which no applicable action exists. Furthermore, suppose that the analysis of this failure yields that the whole search tree following the introduction of A contains no solution. Then, the best reaction with respect to this analysis is to backtrack all actions following A as well as A itself in order to leave this search branch that contains no solutions. Since the dependency directed backtracking component of PLAN can backtrack only one action at time there is no possibility to make use of the available knowledge. PLAN would backtrack A not before

[1] Such simplifications are conducted by actions of the methods SIMPLIFY-F and SIMPLIFY-B. Both methods employ MAPLE to simplify given numerical terms. SIMPLIFY-F is a forward method, which applies MAPLE to the formula of a support in order to derive a new simplified support. SIMPLIFY-B is a backward method, which applies MAPLE to a task in order to reduce the task to a simplified task.

having traversed exhaustively the complete search space following the introduction of A. Thus, when the knowledge is available to backtrack to a certain point in the search space, then it is obviously desirable to backtrack directly sequences of actions at once. In the case studies that are described in chapter 8 and chapter 9, we shall discuss several concrete situations where such knowledge is available.

Another kind of knowledge describes how to productively use failures. For instance, IRELAND and BUNDY describe in [122, 123] how to patch failed proof attempts of the proof planner CLAM by exploiting information on failures. We encountered situations in the limit domain where failures can be productively used. The *Cont-If-Deriv* theorem states that a function f is continuous at point a if it has a derivative f' at point a. In the proof planning process the definition of continuous and derivative in both, the task and the assumption, is replaced first by its ϵ–δ–definition. Further decomposition of the task formula results in a task with formula $|f(c_x) - f(a)| < c_\epsilon$ where c_ϵ and c_x are constants. The decomposition of the assumption results in a new support with formula $|\frac{f(mv_{x'})-f(a)}{mv_{x'}-a} - f'| < mv_{\epsilon'}$ where $mv_{x'}$ and $mv_{\epsilon'}$ are new meta-variables. Indeed, the task can be proved under this assumption. This results — among others — in a task with the formula $mv_{x'} \doteq c_x$, which is closed by an action of the method TELLCS-B that passes the formula to $\mathcal{CoSIE}$. Unfortunately, another task with formula $|mv_{x'} - a| > 0$ is also created during the decomposition of the assumption. This task can be reduced to a task with the formula $mv_{x'} \neq a$. Suppose, we use the information $mv_{x'} \doteq c_x$ by eager instantiation of meta-variables such that this tasks results in $c_x \neq a$. Nevertheless, proof planning reaches a dead end at this task since there is no support available to close it. How can we deal with this failure? The analysis of this and similar situations indicates that a case-split is needed on $c_x \neq a \vee c_x \doteq a$, which has to be introduced before the task $|f(c_x) - f(a)| < c_\epsilon$ is tackled. Then, this task has to be proved for two cases: In the first case, $c_x \neq a$ is assumed and the task $|f(c_x) - f(a)| < c_\epsilon$ can be proved from the assumption as described above. Obviously the problematic subtask $c_x \neq a$ can now be closed directly by the assumption $c_x \neq a$ of the case-split. In the second case, $c_x \doteq a$ is assumed and the task follows since $|f(c_x) - f(a)| < c_\epsilon$ can be simplified to $|f(a) - f(a)| = 0 < c_\epsilon$ by an action of =Subst-B. The resulting task is satisfied by a support with the same formula that resulted from the decomposition of the original task. When should the case-split be introduced? By mathematical intuition it should be introduced when the task $c_x \neq a$ is created and cannot be closed. This demands reasoning about this failure, to backtrack to a certain point in the search space, and to introduce the case-split. An *a priori* introduction of a case-split is not possible since neither the need for a case-split nor the elements for the cases are given.

Another situation where we could make use of failures in a productive way arises in examples like exercise 4.1.3 (see last section). We have to show that

$$\forall \epsilon_1\mathbf{.}(0 < \epsilon_1 \Rightarrow \exists \delta_1\mathbf{.}(0 < \delta_1 \wedge \forall x_1\mathbf{.}(|x_1 - c| > 0 \wedge |x_1 - c| < \delta_1 \Rightarrow |f(x_1) - l| < \epsilon_1)))$$

holds under the assumption that

$$\forall \epsilon\mathbf{.}(0 < \epsilon \Rightarrow \exists \delta\mathbf{.}(0 < \delta \wedge \forall x\mathbf{.}(|x - 0| > 0 \wedge |x - 0| < \delta \Rightarrow |f(x + c) - l| < \epsilon))).$$

The decomposition of the task formula results — among others — in a task with formula $|f(c_{x_1}) - l| < c_{\epsilon_1}$. Unwrapping the assumption yields a new support line with formula $|f(mv_x + c) - l| < mv_\epsilon$. Actually, SOLVE*-B should be applied to this task. However the computation of a corresponding action of this method fails since c_{x_1} and $mv_x + c$ cannot be unified. How can we deal with this failure? We analyzed this situation and similar ones and found that the application of methods is sometimes blocked because unifications of terms do not succeed but have a residue $t_1 = t_2$. For some examples this residue $t_1 = t_2$ is consistent with $\mathcal{C}o\mathcal{SIE}$'s current constraint store. The analysis of these examples indicates that, if (1) a method application is blocked because of a failed unification with a residue $t_1 = t_2$ and (2) $\mathcal{C}o\mathcal{SIE}$ states that this residue $t_1 = t_2$ is consistent with its current constraint store, then we can speculate the lemma $t_1 = t_2$ as new open task and rewrite the task on which the planner failed with this equation. Afterwards the speculated lemma can be closed by an action of TellCS-B and the rewritten task can be solved since the unification becomes unblocked.[2] In our example we would speculate the lemma $mv_x + c \ \doteq \ c_{x_1}$ and would reduce the task with respect to this equation to a new task with formula $|f(c_{x_1}) - l| < mv_\epsilon$. Then, Solve*-B is applicable with respect to the rewritten task and the support $|f(c_{x_1}) - l| < c_{\epsilon_1}$. Similar to the introduction of a case-split, the lemma $t_1 = t_2$ cannot be speculated *a priori*. First, the application of methods such as Solve*-B has to fail. Then, the analysis of this failure can provide suitable t_1 and t_2 such that $t_1 = t_2$ can be speculated.

The lesson learned from these situations and similar ones is that we need different ways to deal with failures and the possibility to reason about a failure in order to flexibly react to it. Moreover, our examples illustrate that the flexible employment of backtracking can be helpful. Although, backtracking should not be the only possibility to react on failures.

6.1.3 Flexible Action Computation and Selection

Similar to the components for backtracking and meta-variable instantiation, also PLAN's action computation and selection cannot be adapted to different problem domains. However, there are situations that need different behaviors.

PLAN uses only the **ChooseAction** subprocedure described in section 4.2.4 to compute and select the next action. **ChooseAction** first selects a method. Then, it chooses with respect to this method supports and parameters and computes all resulting possible actions. Finally, it decides among these actions. If the subprocedure succeeds to find an action for a method, then it will not compute and reason on actions of any other method. An alternative to this subprocedure is a procedure for action selection that computes first all possible actions with respect to all given methods and decides then for an action based on the information of all possible actions. This subprocedure is called **ChooseActionAll**; its pseudo-code description is given in appendix A. The advantage of **ChooseActionAll** is that the decision for one action can be done by control rules based on the knowledge of all possible actions. However, **ChooseActionAll** requires that for all possible methods the matching of method objects with $\mathcal{PDS}$ objects is performed whereas **ChooseAction** avoids these expensive operations as much as possible by checking one method after the other.

Although **ChooseAction** is sufficient for most applications, in some applications the advantages of **ChooseActionAll** outweigh its disadvantages. In [53], we describe the realization of semantically guided proof planning in Ωmega. The idea of semantically guided proof planning (proposed by Choi and Kerber [52]) is to use sets of reference models to guide the choice of the next action to be introduced. The reference models provide a measurement on which actions produce best new

[2] In general, the introduction of unification residues as new tasks opens a Pandora's box: whenever we deal with a residue we introduce some new residues, which in turn must be dealt with. How we restrict the introduction of residues in tasks in order to avoid this problem is described in chapter 8 where we shall discuss the case study on problems from the limit domain.

assumptions or goals. This approach works the better the more actions it can choose from. Thus, **ChooseActionAll** is better suited than **ChooseAction**.

This example is another piece of evidence that we need algorithms that are adaptable to the special needs of different problem classes.

6.1.4 Knowledge of Different Proof Techniques

Mathematicians usually have several proof techniques to tackle a certain class of problems. When analyzed and formalized for proof planning, these proof techniques result in sets of methods and control rules and the knowledge of which sets of methods and control rules belong together becomes part of the domain knowledge of a mathematical domain.

In section 5.1 we introduced the limit domain and described how PLAN constructs ϵ-δ-proofs. PLAN employs a certain set of methods and control rules that prove a limit problem such as $\lim_{x \to 2} x^3 + 2 * x^2 = 16$ with an ϵ–δ–technique. The same problem can be solved also in totally different ways. For instance, based on the basic limit theorems such as LIM+ and LIM*, this problem can also be solved by successively decomposing the function $x^3 + 2 * x^2$ to sums and products for which the theorems can be applied. This proof is shorter and more abstract than the first one and relies on different methods (i.e., methods that make use of already proved facts) and control rules.

As another example, consider the problem to prove that the residue class structure $(\mathbb{Z}_5, \bar{+})$ is associative, which requires to show that for all $x, y, z \in \mathbb{Z}_5$ $x \bar{+} (y \bar{+} z)$ equals $(x \bar{+} y) \bar{+} z$. One proof technique to tackle this problem is to perform an exhaustive case-split on all possible cases of the universally quantified variables that range over finite domains and to check for each single case that the resulting equation holds. Another technique is to reduce the initial task to general equations whose validity is tested, for instance, by a computer algebra system. Again the two techniques employ different sets of methods and control rules and result in different proof plans.

Why is the knowledge of which sets of methods and control rules belong together important for proof planning? To deal with the large sets of methods and control rules that result from the exploration of different mathematical domains is a non-trivial task: if they are employed all at once, then the resulting search space may become unmanageable. However, an *a priori* exclusion of methods and control rules is difficult since doing so may forego the possibility to find the solution. Domain knowledge that describes which sets of methods and control rules belong together can help since it provides a means to structure the large body of methods and control rules.[3]

Connected with the domain knowledge of which methods and control rules form proof technique units is also mathematical knowledge of how to control the combination and application of these units. For instance, there is control knowledge of which unit should be preferred to tackle a particular problem, if several proof techniques for this problem are known. Moreover, there is control knowledge of when

[3] The only existing structuring mechanism for methods and control rules used in the PLAN framework are ΩMEGA's theories in which also methods and control rules are stored. However, methods and control rules that emulate a certain proof technique do not necessarily belong all to the same theory. For instance, to perform ϵ-δ-proofs for limit problems PLAN employs methods that deal with (in)equalities on real numbers (e.g., TellCS-B, ComplexEstimate-B), methods that perform simple manipulations of logical connectives and quantifiers (e.g., $\wedge$I-B, $\forall$I-B), and methods that deal with equations (e.g., =Subst-B). Since these methods are stored in different theories an additional structuring mechanism to group them together is needed to reflect the knowledge of which methods and control rules cooperate to achieve together an ϵ-δ-proof.

a unit should be finished and another one should be started. A switch to another proof technique unit could be caused by the observation that the current proof technique is likely to fail on the given problem and that another proof technique, which seems to be more promising, should be tried. Another reason to switch to another proof technique unit could be that a unit reduces the initial problem to several subproblems for which there are more suitable units. Examples for such control knowledge is given in chapter 9 where we shall discuss the residue structures case study.

PLAN provides no means to employ the described knowledge. This can be provided by an extension of the plain planning that structures methods and control rules and includes meta-reasoning on how to apply and combine the units of methods and control rules.

6.1.5 Knowledge of Parameterized Algorithms and Instances

ΩMEGA provides several components to tackle a theorem, which all refine or modify a $\mathcal{PDS}$. A user of ΩMEGA can choose among proof planning, proof by analogy, and several first-order and higher-order ATPs. Often there is knowledge of which algorithm is suitable to tackle which problems. For instance, the application of the analogy component is sensible only if there is a suitable source problem that has already been proved. First-order ATPs will succeed only if the problem at hand is a first-order problem or can be reduced to a first-order problem. Proof planning is the suitable choice only for problems that belong to domains for which the method and control rule knowledge is available. If the algorithms are parameterized, then the user has to decide which instance of the algorithm to apply (e.g., see [114]).[4]

The knowledge of which instance and algorithm is suitable to tackle which problem is important since it allows for adapting an algorithm to a particular problem. Connected with this knowledge is heuristic knowledge of how to control the combination and application of different instances, e.g., knowledge of how to choose among several applicable instances and algorithms, when to switch to another instance and algorithm, and so on.

PLAN does not allow for a flexible combination of different algorithms for proof refinement and modification and their instances guided by heuristic control knowledge. Its components for action introduction, backtracking, and meta-variable instantiation are connected in a pre-defined way. Algorithms different from these components can be employed by PLAN only within methods and control rules (e.g., ATPs). That is, PLAN does not switch to another algorithm but employs other algorithms only as support systems for proof planning. This forbids, for instance, a combination of proof planning with analogy in which one algorithm passes subproblems to the other algorithm similar to a user who decides for different algorithms and instances in order to tackle different subproblems within one problem solving attempt.

The lesson learned is that we need a mechanism that applies different algorithms and their instances and combines them in one problem solving attempt. The mechanism should be guided by meta-reasoning on how to apply and combine the different algorithms and their instances.

[4] A *parameterized algorithm* provides parameters to determine its behavior. Different *instances of a parameterized algorithm* specify different behaviors of the algorithm by employing different instantiations of its parameters.

6.1.6 Mathematical Experience

The examples described in the preceding sections provide evidence that, in order to tackle heterogeneous sets of problems, different proof plan operations and modifications are necessary that can be flexibly combined guided by meta-reasoning. That is, there is not one proof planning strategy that is suitable for all classes of problems but rather the proof planning approach should be adaptable by meta-reasoning to the needs of different problems.

This observation is consistent with mathematical experience where different problem solving strategies and their flexible applications are crucial human skills in order to adapt the theorem proving to the needs of different classes of problems, as SCHOENFELD points out in his book on mathematical problem solving [209]:

> *As the person begins to work on a problem, it may be the case that some of the heuristic techniques that appear to be appropriate are not. [...] In consequence, having a mastery of individual heuristic strategies is only one component of successful problem solving. Selecting and pursuing the right approaches, recovering from inappropriate choices, [...] is equally important.*
>
> Schoenfeld, 1985, [209] pp. 98–99

Schoenfeld emphasizes the significance of both, the availability of several proof techniques to deal with certain problem classes as well as their controlled application. Several problem solving strategies increase the likelihood that a problem is solved because of several reasons. First, different approaches are necessary to tackle different classes of problems. Second, a pool of approaches for a certain class of problems increases the likelihood that at least one approach can solve a concrete problem from the class. Third, in order to deal with non-trivial mathematical problems it is necessary to tackle different subproblems by different means. Thus, it is necessary to flexibly combine different problem solving strategies and to switch among them during one problem solving process.

Another problem of PLAN, which we discussed in section 6.1.4, is that it provides no means to structure available methods and control rules in meaningful units. For proof planning this is a problem because the search space becomes unmanageable when the number of methods grows and the more control rules the planner has to evaluate the more the proof process may slow down. Again our observation on the need for a structuring mechanism is consistent with mathematical experience. Indeed, categorizing a problem and selecting then the right knowledge to tackle the problem are crucial human skills as SCHOENFELD and HINSLEY , HAYES , and SIMON point out:

> *Individuals with extensive experience in any particular domain categorize their prior experiences in that domain and then use those categorizations both to interpret current situations and to access relevant methods for dealing with those situations.*
>
> Schoenfeld, 1985, [209] p. 244

> *People have a body of information about each problem type which is potentially useful in formulating problems of that type for solution, [...], directing attention to important problem elements, making relevance judgments, retrieving information concerning relevant equations etc.*
>
> Hinsley, Hayes, and Simon, 1977, [115] p. 92

Mathematical knowledge is structured with respect to problem classes to whose solution it can contribute. This avoids a cognitive overload since understanding a problem and recognizing to which problem class it belongs (also called the problem perception in [209]) allows a mathematician to choose the knowledge needed to tackle the problem.

In his book on mathematical problem solving [196] POLYA distinguishes two phases of the knowledge structuring, which he calls *mobilization* and *organization*.

> 1. *In order to solve a problem, we must have some knowledge of the subject-matter and we must select and collect the relevant items of our existing but initially dormant knowledge. [...] Extracting such relevant elements from our memory may be termed mobilization.*
>
> 2. *In order to solve a problem, however, it is not enough to recollect isolated facts, we must combine these facts, and their combination must be well adapted to the problem at hand. [...] This adapting and combining activity may be termed organization.*
>
> Polya, 1971, [196] p. 157

Knowledge-based proof planning provides methods to encode single steps relevant for a certain domain and control rules to combine and adapt the methods. So far, however, it provides no means to encode the result of a mobilization and organization process, i.e., it provides no means to encode which methods and control rules belong together to tackle a certain class of problems.

6.1.7 Summary of Motivation

The examples and scenarios discussed in this section show the main drawbacks of PLAN:

1. PLAN's algorithm cannot be adapted to the particular needs of different classes of problems. Its hard-coded integration of very restricted components for action introduction, backtracking, and meta-variable instantiation represents just one particular problem solving strategy suitable for many problems of the limit domain but insufficient as a general technique.

2. The combination with other algorithms that can contribute to the solution of a proof planning problem is not sufficiently supported.

3. A lot of domain knowledge of different proof plan refinements and modifications and their combination is available. However, since this knowledge is beyond the capabilities of methods and control rules, there is no means to incorporate and use it in PLAN.

Our examples illustrate that, in order to tackle heterogeneous sets of problems, various plan refinements and modifications are necessary. In particular, in order to enable different problem solving behaviors and the flexible adaption to the needs of different (sub)problems, the decision on when to call a certain refinement and modification should not be encoded once and forever into the system but rather be determined by meta-level reasoning using available heuristic control knowledge.

6.2 The Concepts of Multi

From the observation of the drawbacks of PLAN (see the previous section) we derive the following requirements for the design of the new system Multi:

- In Multi, the planning functionalities meta-variable instantiation, backtracking, and action introduction should be clearly separated algorithms.
- Multi should enable the incorporation of other algorithms that can contribute to the proof plan construction.
- Multi should allow for the specification and incorporation of different instances of employed parameterized algorithms.
- Multi should provide a structuring mechanism for methods and control rules.
- Multi should enable the combination of the different algorithms and their instances within one problem solving approach.
- In Multi, the decision on when to call a certain algorithm or instance should not be hard-coded into the system but rather be determined by meta-level reasoning using available heuristic control knowledge.

In order to meet these requirements, proof planning with multiple strategies in Multi decomposes the previous monolithic proof planning process and replaces it by separated parameterized algorithms as well as different instances of these algorithms, so-called strategies. The strategies, which specify different behaviors of the algorithms, are the basic elements for proof construction in multiple-strategy proof planning. That is, the goal of multiple-strategy proof planning is to compute a sequence of strategy applications that derives a given theorem from a given set of assumptions. The decision on when to apply a strategy is not encoded once and forever into the system but rather is determined by meta-level reasoning using heuristic control knowledge of strategies and their combination.

In the following, we first introduce in section 6.2.1 the basic concepts of proof planning with multiple strategies and illustrate them with examples. Then, we describe in section 6.2.2 Multi's blackboard architecture. Section 6.2.3 discusses the reasoning at the strategy-level with strategic control rules. We conclude with an informal description of all algorithms currently employed by Multi that are not exemplified in section 6.2.1.

6.2.1 Algorithms, Strategies, and Tasks

Algorithms

Multi enables the incorporation of heterogeneous, parameterized algorithms for different kinds of proof plan refinements and modifications. Currently, Multi employs the following algorithms (technical descriptions of these algorithms, i.e., of the plan refinements or modifications they perform, are given in chapter 7):

PPlanner refines a proof plan by introducing new actions.

InstMeta refines a proof plan by instantiating meta-variables.

BackTrack modifies a proof plan by removing refinements of other algorithms.

Exp refines a proof plan by expanding complex steps.

ATP refines a proof plan by solving subproblems with machine-oriented automated theorem provers.

CPLANNER refines a proof plan by transferring steps from a source proof plan or fragment.

The decomposition of the previous monolithic proof planner of ΩMEGA allows to extend and generalize the functionalities of its subcomponents. This results in the independent and parameterized algorithms **PPLANNER**, **INSTMETA**, and **BACKTRACK** for action introduction, meta-variable instantiation, and backtracking. **EXP**, **ATP**, and **CPLANNER** integrate new refinements of the proof plan.

Strategies

Instances of these algorithms can be specified in different strategies. Technically, a *strategy* is a condition-action pair. The condition part states when the strategy is applicable. The action part consists of a modification or refinement algorithm and an instantiation of its parameters. Similar to the knowledge of the applicability of methods we separate the legal and heuristic knowledge of the applicability of strategies. The condition part of a strategy states the legal conditions that have to be satisfied in order for the strategy to be applicable, whereas *strategic control rules* reason about the heuristic utility of the application of strategies.

To execute or to apply a strategy means to apply its algorithm to the current proof planning state with respect to the parameter instantiation specified by the strategy. For instance, the parameters of **PPLANNER** are a set of methods, a list of control rules, a termination condition, and an action selection procedure. When MULTI executes a **PPLANNER** strategy, the **PPLANNER** algorithm introduces only actions that use the methods specified in the strategy. The actions are computed and selected by the action selection procedure (e.g., **CHOOSEACTION** or **CHOOSEACTIONALL**) specified by the strategy. The action selection procedures evaluate then the control rules specified by the strategy during the computation of actions. The application of the strategy terminates, when its termination condition is satisfied. Hence, different strategies of **PPLANNER** provide a means to structure the method and control rule knowledge. Both algorithms, **INSTMETA** and **BACKTRACK**, have one parameter. The parameter of **INSTMETA** is a function that determines how the instantiation for a meta-variable is computed. If MULTI applies a **INSTMETA** strategy with respect to a meta-variable mv, and if the computation function of the strategy yields a term t for mv, then **INSTMETA** substitutes mv by t in the proof plan. The parameter of **BACKTRACK** is a function that computes a set of refinement steps of other algorithms that have to be deleted. When MULTI applies a **BACKTRACK** strategy, then **BACKTRACK** removes all refinement steps that are computed by the function of the strategy as well as all steps that depend from these steps.

Notation 6.1: Strategies are denoted in the sans serif font (e.g., NormalizeLineTask, UnwrapHyp).

Tasks

MULTI extends the task concept of PLAN. Since MULTI employs further kinds of tasks, the tasks used in PLAN (i.e., a pair consisting of an open line and its supports) are called *line-tasks* in MULTI. MULTI uses also *instantiation-tasks* and *expansion-tasks*. The introduction of a meta-variable into the plan results in an instantiation-task, that is, the task to instantiate this meta-variable. Similarly, the introduction of a method or tactic step into the $\mathcal{PDS}$, which is constructed during the proof planning process, results in an expansion-task, that is, the task to expand this step. An instantiation-task stores the meta-variable for which an instantiation has to be constructed. The instantiation task for meta-variable mv is written as $mv|^{Inst}$. An expansion-task consists of a proof line L in the $\mathcal{PDS}$, which is justified with a method or a tactic application. The expansion-task with line L is written as $L|^{Exp}$. MULTI stores all used kinds of tasks in an agenda.

Different tasks can be tackled by different algorithms and strategies. For instance, since strategies of **INSTMETA** introduce instantiations for meta-variables they are suitable to tackle instantiation-tasks. **EXP** is the suitable choice to deal with expansion-tasks, whereas strategies of **PPLANNER** or **ATP** can tackle line-tasks. A strategy checks in its condition part whether it is applicable to a particular task. That is, the condition of a strategy is a predicate on tasks. To *apply a strategy to a task* means to execute the strategy with respect to the task.

The algorithms and kinds of tasks currently employed by MULTI have been derived from the case studies. However, the MULTI framework is envisaged to be extended by further algorithms and further kinds of tasks, if needed.

Example Strategies

In the following, we describe some strategies needed to accomplish ϵ-δ-proofs (see section 5.1). The methods and control rules for ϵ-δ-proofs are structured into the three strategies NormalizeLineTask, UnwrapHyp, and SolveInequality. All three strategies are instantiations of **PPLANNER**. A more detailed description of the application of these strategies and their cooperation when accomplishing ϵ-δ-proofs is given in section 8.1.

The strategy SolveInequality (see Table 6.1) is applicable to prove line-tasks whose formulas are inequalities or whose formulas can be reduced to inequalities. It comprises methods such as COMPLEXESTIMATE-B, TELLCS-B, TELLCS-F, ASKCS-B, and SOLVE*-B (see section 5.1). Its list of control rules contains the rules `prove-inequality` and `eager-instantiate`. Possible actions are computed and selected with the **CHOOSEACTION** procedure. The strategy terminates, when there are no further line-tasks whose formulas are inequalities or whose formulas can be reduced to inequalities. Note that it is the parameterization of **PPLANNER** that makes SolveInequality appropriate to tackle line-tasks whose formulas are inequalities as stated in the condition part of the strategy.

Strategy: SolveInequality		
Condition	*inequality-task*	
Action	Algorithm	**PPLANNER**
	Action Procedure	**CHOOSEACTION**
	Methods	COMPLEXESTIMATE-B, TELLCS-B, TELLCS-F, SOLVE*-B, ASKCS-B ...
	C-Rules	`prove-inequality`, `eager-instantiate`, ...
	Termination	*no-inequalities*

Table 6.1: The SolveInequality strategy.

NormalizeLineTask (see Table 6.2) is used to decompose line-tasks whose goals are complex formulas with logical connectives and quantifiers. Typical methods in NormalizeLineTask are $\wedge$I-B and $\forall$I-B (see section 5.1). NormalizeLineTask employs the **CHOOSEACTION** procedure for the action computation and selection and terminates, when all complex line-tasks are decomposed to literal line-tasks.

The aim of UnwrapHyp (see Table 6.3) is to unwrap a focused subformula of an assumption in order to make it available for proving a line-task. The list of its

Strategy: NormalizeLineTask		
Condition	*complex-line-task*	
Action	Algorithm	**PPLANNER**
	Action Procedure	**CHOOSEACTION**
	Methods	$\forall$I-B, $\exists$I-B, $\wedge$I-B, ...
	C-Rules	
	Termination	*literal-line-tasks-only*

Table 6.2: The NormalizeLineTask strategy.

methods includes, for instance, $\forall$E-F and $\wedge$E-F. The control rule `tackle-focus` determines that, if UnwrapHyp is applied, then the actions of the available methods can be used only if they use a support in their premises that carries a focus and when their conclusions do not tackle the focused subformula. For instance, if a line-task has the supports $B_1 \wedge B_2$ and $A_1 \wedge (A_2 \wedge \mathit{focus}(A_3 \wedge A_4))$, then only actions of $\wedge$E-F that use the second support with the focus are allowed. The introduction of two actions of $\wedge$E-F derive the new support $\mathit{focus}(A_3 \wedge A_4)$ to which no further action of $\wedge$E-F can be applied since it would decompose the focused subformula. Similar to NormalizeLineTask and SolveInequality, UnwrapHyp uses the **CHOOSEACTION** algorithm. It terminates as soon as all focused formulas are unwrapped.

Strategy: UnwrapHyp		
Condition	*focus-in-subformula*	
Action	Algorithm	**PPLANNER**
	Action Procedure	**CHOOSEACTION**
	Methods	$\forall$E-F, $\exists$E-F, $\wedge$E-F, ...
	C-Rules	`tackle-focus`
	Termination	*focus-at-top*

Table 6.3: The UnwrapHyp strategy.

In order to instantiate meta-variables that occur in constraints collected by $\mathcal{C}o\mathcal{SIE}$, we implemented the two **INSTMETA** strategies InstIfDetermined and ComputeInstFromCS (see Table 6.4). InstIfDetermined is applicable only, if $\mathcal{C}o\mathcal{SIE}$ states that a meta-variable is already determined by the constraints collected so far. Then, the computation function connects to $\mathcal{C}o\mathcal{SIE}$ and receives this unique instantiation for the meta-variable. ComputeInstFromCS is applicable to all meta-variables for which constraints are stored in $\mathcal{C}o\mathcal{SIE}$. The computation function of this strategy requests from $\mathcal{C}o\mathcal{SIE}$ to compute an instantiation for a meta-variable that is consistent with all constraints collected so far.

Strategy: InstIfDetermined		
Condition	*determined-in-cs*	
Action	Algorithm	**INSTMETA**
	Function	*get-determined-instantiation*

Strategy: ComputeInstFromCS		
Condition	*mv-in-cs*	
Action	Algorithm	**INSTMETA**
	Function	*compute-consistent-instantiation*

Table 6.4: The **INSTMETA** strategies InstIfDetermined and ComputeInstFromCS.

The dependency-directed backtracking described in section 4.2.3 is realized as the strategy BackTrackActionToTask (see Table 6.5) of the **BACKTRACK** algorithm. BackTrackActionToTask instantiates the **BACKTRACK** algorithm with the function *step-to-line-task*, which computes the action that introduced a line-task. BackTrackActionToTask is applicable to each line-task.

Strategy: BackTrackActionToTask		
Condition	*line-task*	
Action	Algorithm	**BACKTRACK**
	Function	*step-to-line-task*

Table 6.5: The BackTrackActionToTask strategy.

6.2.2 MULTI's Blackboard Architecture

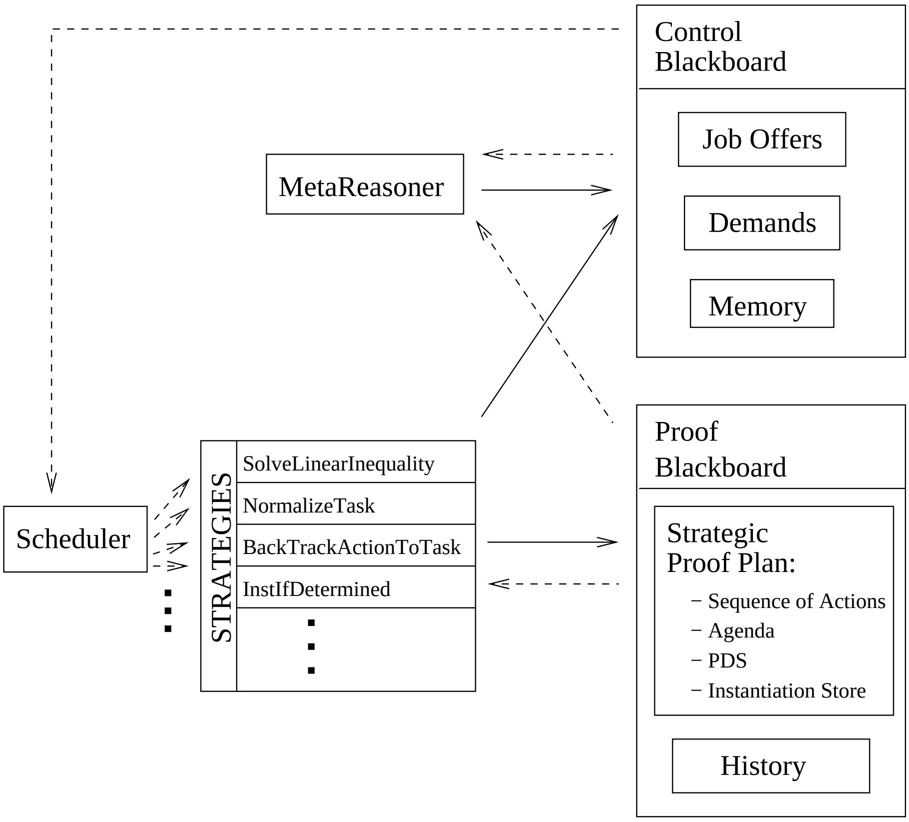


Figure 6.1: MULTI's blackboard architecture.

When we designed proof planning with multiple strategies, we aimed at a system that allows for the flexible cooperation of independent components for proof plan refinement and modification, guided by meta-reasoning. For the implementation we decided to use a blackboard architecture because this is an established means to organize the cooperation of several independent components, so-called knowledge sources, for solving a complex problem. Blackboard systems do not rely on a pre-defined control of the application of the involved components but provide the flexibility to employ their knowledge sources opportunistically as the following quotations point out:

> *As we hope to illustrate in this book, the blackboard model is a very simple yet powerful idea for coping with problems characterized by the need to deal with [...] a non-deterministic solution strategy.*
>
> Engelmore and Morgan, 1988, [76] Preface pix

> *As a result, the sequence of knowledge source invocation is dynamic and opportunistic rather than fixed and pre-programmed.*
>
> Engelmore and Morgan, 1988, [76] p14

In the following, we give an informal overview on MULTI and the ideas behind it. A detailed technical description of the algorithms and concepts as well as a formal definition of strategic proof planning with MULTI are given in the next chapter.

MULTI's architecture is displayed in Figure 6.1. In this figure dashed arrows indicate information flow whereas solid arrows indicate that a knowledge source changes the content of the respective blackboard. MULTI's architecture is similar to the HEARSAY-III and the BB1 blackboard systems, which we discussed in section 2.2, in that it employs two blackboards, the so-called *proof blackboard* and the *control blackboard*.

We decided for a two-blackboard architecture to emphasize the importance of both the solution of the proof planning problem whose status is stored on the proof blackboard and the solution of the control problem, that is, which possible strategy should the system perform next. Moreover, the two blackboard architecture is more suitable for potential extensions of our approach that we shall discuss in section 6.3 and section 6.4. The proof blackboard contains the current strategic proof plan, which consists of a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, which store the collected instantiations of meta-variables, as well as the strategic history. The control blackboard contains three repositories to store information relevant for the control problem: job offers, demands, and a memory.

Corresponding to the two blackboards, there are also two sets of knowledge sources shown in Figure 6.1 that work on these blackboards. The strategies are the knowledge sources that work on the proof blackboard. A strategy can change the proof blackboard by refining or modifying the agenda, the $\mathcal{PDS}$, the history of strategies, and bindings of the meta-variables. The strategy component contains all the strategies that can be used. If a strategy's condition part is satisfied with respect to a certain task in the agenda, then the strategy posts its applicability with respect to this task as a job offer onto the control blackboard. Technically, a *job offer* is a pair (S,T) with a strategy S and a task T, which signs that T satisfies the condition of S. That is, in the terminology of blackboard systems, a task that satisfies the condition of a strategy is the event that triggers the strategy. The MetaReasoner is the knowledge source working on the control blackboard. It evaluates strategic control knowledge represented by strategic control rules in order to rank the job offers. The architecture contains a *scheduler* that checks the control blackboard, for its highest ranked job offer. Then, it executes the strategy of the job offer with respect to the task specified in the job offer. In a nutshell, MULTI operates according to the cycle in Figure 6.2, which passes the following steps:

Job Offer Strategies whose condition is true put a job offer onto the control blackboard.

Guidance The MetaReasoner evaluates the strategic control rules to order the job offers on the control blackboard.

Invocation A scheduler invokes the strategy who posed the highest ranked job offer.

Execution The algorithm of the invoked strategy is executed with respect to the parameter instantiation specified by the strategy.

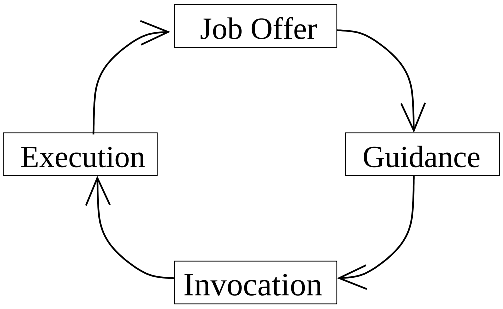


Figure 6.2: Cycle of Multi.

The choice of a job offer can depend on particular demand information issued by strategies onto the control blackboard and the content of the memory. An executed strategy can reason on whether it should interrupt. This can be sensible if the strategy is stuck or if it turns out that it should not proceed before another strategy is executed. Then, the execution of a strategy interrupts itself, places demands for other strategies onto the control blackboard, and stores a pair consisting of its execution status and the demands it posed in the memory. Interrupted executions of a strategy stored in the memory place job offers for their re-invocation onto the control blackboard. A job offer from the memory consists just of a pointer to the memory entry that posed this job offer. If such a job offer is scheduled, the interrupted strategy execution is re-invoked from the memory.

By posing demands and interrupting strategies particularly desired cooperations between strategies can be realized. For instance, we discussed in section 6.1.1 that certain problems on which PLAN fails could be solved if meta-variables would be instantiated as soon as $\mathcal{C}o\mathcal{SIE}$ states that they are uniquely determined. In order to realize this the **InstMeta** strategy InstIfDetermined and the **PPlanner** strategy SolveInequality have to cooperate. This cooperation works as follows: The strategy SolveInequality contains the control rule `eager-instantiate`. If evaluated during an execution of SolveInequality, this control rule checks whether InstIfDetermined is applicable for an occurring meta-variable. If this is the case, it causes the interruption[5] of the execution of the SolveInequality strategy and poses the demand that InstIfDetermined should be applied with respect to the instantiation-task of the meta-variable. The status of the interrupted SolveInequality strategy is stored in the memory from where it can be reinvoked as soon as the posed demand is satisfied by the corresponding application of InstIfDetermined.

6.2.3 Reasoning at the Strategy-Level

In the Multi system, no order or combination of refinements or modifications on the proof blackboard is pre-defined. The choice of strategy applications results from meta-reasoning at the strategy-level that is conducted by the MetaReasoner, which evaluates the strategic control rules on the job offers on the control blackboard. Strategic control rules are formulated in the same control rule language as control rules on tasks, methods, supports and parameters, and actions (see section 4.1.3). They can reason about all information stored on the control blackboard and the

[5] Interruption is an explicit choice point in the **PPlanner** algorithm, see section 7.5.2.

proof blackboard (i.e., about the proof plan constructed so far and the plan process history) as well as about the mathematical domain of the proof planning problem.

The advantage of this knowledge-based control approach is that the control of Multi can be easily extended and changed by modifying the strategic control rules. In contrast, when the combination of integrated components of a system is hard-coded into a control procedure, then each extension or change requires reimplementation of parts of the main control procedure. Moreover, the strategic control rules declaratively represent the heuristical control knowledge of how to combine the strategies of Multi, so that this knowledge can be communicated to the user.

In the following, we shall discuss five strategic control rules, which are the backbone of the strategic control in Multi.

```
(control-rule prefer-demand-satisfying-offers
              (kind strategic)
              (IF (job-offer-satisfies-demand JO))
              (THEN (prefer JO)))

(control-rule prefer-memory-offers
              (kind strategic)
              (IF (and (job-offer-from-memory JO)
                       (no-further-demands JO)))
              (THEN (prefer JO)))

(control-rule defer-memory-offers
              (kind strategic)
              (IF (and (job-offer-from-memory JO)
                       (further-demands JO)))
              (THEN (defer JO)))
```

Figure 6.3: The three strategic control rules `prefer-demand-satisfying-offers`, `prefer-memory-offers`, and `defer-memory-offers`.

The use of demands and the memory for the goal-directed cooperation of strategies is realized by the strategic control rules `prefer-demand-satisfying-offers`, `prefer-memory-offers`, and `defer-memory-offers` given in Figure 6.3. The rule `prefer-demand-satisfying-offers` states that, if a job offer on the control blackboard satisfies a demand on the control blackboard, then this job offer is preferred. Similarly, `prefer-memory-offers` states that, if there is a job offer from an interrupted strategy execution in the memory and all demands of this strategy execution are already satisfied, then this job offer should be preferred. `defer-memory-offers` defers job offers from interrupted strategy executions, if they have still unsatisfied demands.

The rules `prefer-backtrack-if-failure` and `reject-applied-offers` (see Figure 6.4) realize a basic failure reasoning and the rejection of already applied strategies. The purpose of the `prefer-backtrack-if-failure` rule is to integrate backtracking with strategies of **PPlanner**. When a **PPlanner** strategy runs into a failure, that is, it encounters a line-task for which it finds no applicable action, then it interrupts and stores the status of its execution in the memory. `prefer-backtrack-if-failure` causes backtracking by preferring a job offer of the BackTrackActionToTask strategy with the line-task on which the execution of the **PPlanner** strategy failed. Afterwards, the interrupted strategy ex-

```
(control-rule reject-applied-offers
              (kind strategic)
              (IF (job-offer-already-applied JO))
              (THEN (reject JO)))

(control-rule prefer-backtrack-if-failure
              (kind strategic)
              (IF (and (algorithm-of-last-strategy-is PPLANNER)
                       (last-strategy-failure-on-line-task T)
                       (backtrack-job-offer-on JO T)))
              (THEN (prefer JO)))
```

Figure 6.4: The strategic control rules `reject-applied-offers` and `prefer-back-track-if-failure`.

ecution can be re-invoked on the changed proof blackboard. The idea behind `reject-applied-offers` is that a strategy that failed on a task should not be tried again on this task (although it is still applicable to the task, and, thus, it places a job offer onto the control blackboard). `reject-applied-offers` checks whether a job offer corresponds to a strategy execution that has already been tried but was backtracked later on. In this case, `reject-applied-offers` rejects the job offer.

The priority[6] of these control rules increases in the following order: `prefer-demand-satisfying-offers`, `prefer-memory-offers`, `defer-memory-offers`, `reject-applied-offers`, `prefer-backtrack-if-failure`. Although these control rules are the backbone of Multi's control, they realize only a default behavior and can be excluded by the user of Multi or can be overridden by other strategic control rules with higher priority. For instance, in the case studies in chapter 8 and chapter 9 we shall see how more specific control rules enable an elaborate failure reasoning or cause the repeated application of the same strategy although it failed several times on a task.

6.2.4 Further Algorithms

The strategies **PPlanner**, **InstMeta**, and **BackTrack** are introduced and exemplified in section 6.2.1. Here we shall informally introduce the other three algorithms used in Multi, namely **Exp**, **ATP**, and **CPlanner**. Formal descriptions of all algorithms can be found in section 7.5 in the next chapter.

Exp

The algorithm **Exp** tackles expansion-tasks. An expansion-task does not refer directly to an introduced action but contains a proof line in the constructed $\mathcal{PDS}$ whose justification is a complex step, that is, a method or a tactic application. For a proof line L with an abstract justification $(J\ P_1\ \ldots P_n)$ where J is a method or a tactic and $P_1, \ldots, P_n$ are the premises, **Exp** computes a proof segment, which derives L from $P_1, \ldots, P_n$ at a lower level of abstraction. If J is a method, then **Exp** computes the proof segment by instantiating the proof schema of J. If J is a tactic, then **Exp** evaluates the expansion function of J. Afterwards, **Exp** adds the

[6] The MetaReasoner evaluates first the strategic control rules with lower priority. Since they are evaluated later on, the strategic control rules with higher priority cause the final changes of the alternative list of job offers.

new proof lines into the constructed $\mathcal{PDS}$ and adds a new justification to L at a lower level of abstraction.

Currently, the algorithm **EXP** is not parameterized. Since we distinguish technically between a strategy and its algorithm we have implemented the strategy ExpS as the only strategy for the **EXP** algorithm. The application condition of ExpS states that this strategy is applicable to all expansion-tasks.

ATP

The algorithm **ATP** enables the application of automated theorem provers within MULTI in order to prove line-tasks. Its parameters are two functions for the application of an automated theorem prover (or several ones) and the check whether the obtained output is accepted as a proof. The first function obtains as input the line-task to which the **ATP** strategy is applied and returns the output of the employed ATP(s). The second function obtains the output of the ATPs and returns either *true* or *false* where true means that the function accepts the output as proof.

When a strategy of **ATP** succeeds for a line-task $L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$, then **ATP** closes the line L_{open} by the application of the tactic *atp* to the premises $\mathcal{SUPPS}_{L_{open}}$. Moreover, the output obtained from the application function of the strategy becomes the parameter of the justification. Whether this tactic application can be expanded depends on the accepted output. Currently, the expansion function of *atp* can deal with the following outputs:

- Resolution proofs from the provers OTTER [150], BLIKSEM [68], SPASS [239], PROTEIN [13], and equational proofs produced by the provers EQP [152] and WALDMEISTER [114]. On these outputs the expansion function of *atp* calls TRAMP [159], a proof transformation system that transforms resolution-style proofs into assertion level ND-proofs to be integrated into the $\mathcal{PDS}$.
- ND-proofs produced by TRAMP, if TRAMP is used as prover and not as transformation system (see below), and — with little transformational effort — ND-proofs provided by the higher-order prover TPS [8] (see [16] on what kind of transformations are necessary to incorporate TPS proofs into a $\mathcal{PDS}$).

Other output of automated theorem provers can be accepted by the respective strategies of **ATP** but cannot be further processed currently by the expansion function of the *atp* tactic.

Strategy: CallTramp		
Condition	*first-order-problem*	
Action	ATP Apply	*employ-tramp-on-task*
	ATP Output Check	*check-assertion-proof*

Table 6.6: The CallTramp strategy.

As example of a strategy of **ATP** consider CallTramp, which is depicted in Table 6.6. The application condition of CallTramp, *first-order-problem*, is satisfied by line-tasks, whose formulas are first-order. The application function, *employ-tramp-on-task*, employs TRAMP not as transformation module but as prover. This is possible since TRAMP cannot only transform the output of the connected provers but can also call these provers on a problem. When employed in this mode, TRAMP obtains a problem formalization, calls the connected automated theorem provers on the problem, and returns — if one of the connected provers succeeds — an assertion-level

ND-proof. The output check function of CallTramp, *check-assertion-proof*, checks whether the output provided by TRAMP is an ND-proof of the task.[7]

CPLANNER

Case-based reasoning is the approach to tackle new problems or subproblems by adapting given solutions or parts of given solutions of other problems or subproblems [47]. Case-based reasoning components for ΩMEGA were first developed as stand-alone systems not directly intertwined with the proof planner or other components. The last system developed in this paradigm was the TOPAL system [231, 173].

TOPAL obtains as input a *source proof plan* and a *target problem*. It successively transfers method applications from the source proof plan into a proof plan of the target problem. To do so, it computes and maintains possible mappings from objects of the source proof plan (e.g., tasks and proof lines) to corresponding objects of the target proof plan. With these mappings it computes new actions for the target proof plan from actions in the source proof plan. TOPAL processes the given source proof plan chronologically which means that TOPAL selects the actions to transfer in the order of the source proof plan.

The **CPLANNER** algorithm in MULTI extends TOPAL in several ways. First, **CPLANNER** is parameterized and enables the realization of different kinds of case-based reasoning. For instance, we realized a task-directed approach as an alternative to the chronological TOPAL approach. This task-directed approach, which is encoded in the **CPLANNER** strategy TaskDirectedAnalogy (see Table 6.7), first selects a task in the target proof plan and then selects an action to transfer in the source proof plan depending on the selected task. Second, **CPLANNER** allows not only for the transfer of method applications but also for the transfer of strategy applications from a strategic source proof plan into a strategic target proof plan. Moreover, the integration of **CPLANNER** into MULTI enables the flexible combination of case-based reasoning with the other algorithms in MULTI.

The parameters of **CPLANNER** are a list of so-called *action transfer procedures*, a list of control rules, and a termination condition. Action transfer procedures describe how source actions are transfered into target actions. The control rules guide the selection of action transfer procedures and interrupts. The termination condition specifies when the execution of the strategy terminates.

Technically, an action transfer procedure is a triple of a list of choice points, a list of instantiation functions, and a computation function. The choice points specify which objects have to be selected during the transfer process, the instantiation functions provide the alternative lists for the choice points, respectively, and the computation function computes either a new target action or a new demand for a tuple of selected objects. When the computation function provides a new target action, then **CPLANNER** introduces this action into the proof plan under construction. A demand causes **CPLANNER** to interrupt with this demand (see section 7.5.3 for details).

For instance, $TaskMeth$ is an action transfer procedure that realizes a task-directed transfer of source actions. $TaskMeth$ specifies the choice points target task, source action, target premises, and target parameters in this order. That is, it first selects the task in the target problem to tackle and then selects the action to transfer in the source problem depending on this task. Finally, it chooses the target premises and target parameters depending on the selected target task and

[7] *check-assertion-proof* checks only whether the returned object is a proof tree whose root is the goal of the task and whose leaves are the supports of the task. It does not check whether each justification is correct since this would demand to expand the assertion-level proof.

the selected source action. The computation function of *TaskMeth* obtains the chosen objects as input and computes a new action of the method of the source action.

TaskInst is an action transfer procedure for applications of **InstMeta** strategies. It first chooses an instantiation-task in the target plan. Next, it chooses an application of an **InstMeta** strategy in the source plan. Then, its computation function creates the demand to tackle the instantiation-task with the **InstMeta** strategy of the source action.

TaskPPlanner is an action transfer procedure for applications of **PPlanner** strategies. *TaskPPlanner* first chooses a line-task in the target proof plan and next an application of a **PPlanner** strategy in the source plan. The application of a **PPlanner** strategy essentially consists of a sequence of method actions (see section 7.2 for details). *TaskPPlanner* reduces the transfer of the selected **PPlanner** strategy application to the transfer of the corresponding method action sequence. That is, it creates a demand for the recursive application of the **CPlanner** strategy TaskDirectedAnalogy with respect to the selected task and with the sequence of method actions as source actions.

The action transfer procedures *TaskMeth*, *TaskInst*, and *TaskPPlanner* are combined in the **CPlanner** strategy TaskDirectedAnalogy, which is given in Table 6.7, in order to realize the task-directed transfer approach. The application condition of TaskDirectedAnalogy, *always-true-line+inst*, is satisfied by all line- and instantiation-tasks. The list of control rules is empty. The termination condition, *no-local-tasks*, is satisfied, when the initial task to which the strategy is applied and all tasks derived from this task are closed.

Strategy: TaskDirectedAnalogy		
Condition	*always-true-line+inst*	
Action	Action Trans. Procs.	*TaskMeth*, *TaskPPlanner*, *TaskInst*
	C-Rules	$\emptyset$
	Termination	*no-local-tasks*
	Source Actions	(free)

Table 6.7: The TaskDirectedAnalogy strategy

The applicability of TaskDirectedAnalogy is not only restricted by its condition *always-true-line+inst*, but also by its additional parameter, source actions, which is not a parameter of the algorithm **CPlanner**. Such additional parameters of strategies are called *free parameters*. They are not instantiated once and forever in the strategy. Rather, strategic control rules can suggest instantiations for a free parameter during the proof planning attempt.[8] A strategy with free parameters is applied only if a strategic control rule instantiates the free parameters.

The free parameter of TaskDirectedAnalogy, source actions, has to be instantiated by a strategic control rule with the sequence of source actions that the strategy should transfer.[9] A strategic control rule can choose, for instance, a complete source proof plan from a database of solved problems or it can choose a subsequence of actions of a given source proof plan. Instead of using actions of another problem (so-called *external analogy*) a strategic control rule can also suggest a subsequence

[8] Technically, strategies with free parameters post job offers, when their condition is satisfied and strategic control rules can then instantiate the free parameters by attaching instantiations to the job offer.

[9] The instantiation functions of the action transfer procedures look up the given source actions during the execution of the strategy and suggest then alternatives depending on these actions.

of actions of the proof plan under construction to be transfered to another part of the same proof plan (so-called *internal analogy*).

Examples for the application of the TaskDirectedAnalogy strategy in the case studies are given in section 8.2.1. Further examples and a detailed discussion of case-based reasoning in Multi can be found in [210].

6.3 Discussion of the Architecture

In the previous section, we gave reasons for our decision to realize Multi as a blackboard. In this section, we shall discuss how Multi's blackboard architecture compares to other existing blackboard architectures. In particular, we shall compare Multi's architecture with the two blackboard frameworks BB1 and Hearsay-III and point out possible extensions for Multi similar to features of BB1 and Hearsay-III. Afterwards, we shall discuss how the strategies in Multi compare to standard concepts of agents and why we did not implement a multi-agent architecture for Multi. We conclude with a brief discussion of the fundamental differences between Multi and ΩAnts, the other blackboard-based component in Ωmega.

6.3.1 Blackboard Architectures

We start with a discussion of some general features of Multi that relate it to several of the classical blackboard architectures as, for instance, discussed in [76] (see also section 2.2). Afterwards, we compare it with the BB1 and the Hearsay-III blackboard architectures (see section 2.2.2 and section 2.2.3).

6.3.1.1 General Discussion of Multi's Architecture

Knowledge Sources

Multi has two different kinds of knowledge sources: the strategies and the MetaReasoner. The strategies are condition-action pairs, which is a well-established approach in blackboard systems used already in the Hearsay-II [77] system. In contrast, the MetaReasoner evaluates sets of strategic control rules and is comparable with the knowledge sources of the Hasp [181] system, which are sets of rules.

Hierarchies

It is a well-established approach for blackboard systems to organize the blackboards as well as the knowledge sources hierarchically. Some knowledge sources work only at one particular hierarchy level, whereas other knowledge sources transfer information from one level to other levels. For instance, the Hearsay-II system, which is used for speech recognition, distinguishes the phrase-level and the word-level. There are knowledge sources that work on the entries of one level only, respectively, and there are knowledge sources that produce phrase-level entries based on existing word-level entries. Multi employs two blackboards, which are both divided into regions. However, there is no hierarchy relation between these regions. Rather, they just separate different kinds of information. Moreover, a knowledge source in Multi is not associated with a certain region on the blackboard but can change several parts simultaneously.

Parallel vs. Sequential

The use of multiple, independent sources of knowledge enables the exploitation of parallel programming techniques. Examples for blackboard-based approaches that enable parallelism are the CAGE [180] and the POLIGON [180, 201] system. In [180] NII *et al.* describe different ways to exploit parallelism in blackboard systems. In particular, they mention the concurrent application of different knowledge sources and the concurrency of processes within knowledge sources. They also describe problems originating from concurrency. If knowledge sources work concurrently, then each knowledge source has to be able to write on the blackboard without hindering other running knowledge sources or knowledge sources scheduled for execution. Hence, systems whose blackboards and knowledge sources are hierarchically arranged are particularly suited to exploit concurrency since knowledge sources that work at different levels of the blackboard can always be applied concurrently without hindering each other.

In the current implementation, MULTI does not exploit concurrency for two reasons. First, in MULTI there are no different levels or parts of the blackboards on which knowledge sources could easily work concurrently. Second, strategies are often connected in complex ways which complicates their concurrent execution.

For instance, consider a proof situation, where a line-task is tackled by a strategy S_P of **PPLANNER** and an instantiation-task is tackled by a strategy S_I of **INSTMETA**. Potential actions of S_P may depend on the execution of S_I. That is, whether or not S_I does instantiate the meta-variable of the instantiation-task enables or disables actions in **PPLANNER**. If S_P and S_I are executed concurrently, then the success of S_P may depend on the arbitrary moment of the instantiation. As another example consider two line-tasks, which are tackled by two strategies S_1 and S_2 of **PPLANNER** that pass constraints to $\mathcal{C}o\mathcal{SIE}$ (e.g., two executions of the strategy SolveInequality). It is possible that S_1 fails when executed after S_2. This happens if constraints passed by S_1 are inconsistent with constraints, which were passed by S_2 and were already accepted by $\mathcal{C}o\mathcal{SIE}$. If S_1 is executed first and S_2 is executed second, then S_2 may succeed by introducing other actions although $\mathcal{C}o\mathcal{SIE}$ might reject some passed constraints. If S_1 and S_2 are executed concurrently, then the success of strategy S_1 may depend on the random order in which both strategies pass their constraints. In both situations the success of concurrently executed strategies may depend on the actual order of particular operations. Since we want to avoid such random effects influencing the solution process we prefer the sequential execution of strategies explicitly guided by the control knowledge in control rules in MULTI (e.g., control rules that perform a certain meta-variable instantiation at a certain moment).

A potential way to exploit parallelism in MULTI could be to concurrently apply several strategies to the same task, if several job offers for one task are ranked equally good by the strategic control rules. This would allow to check the performance of several strategies in a competitive manner rather than to apply them sequentially and recover from failing ones. We have not realized this approach so far, since it requires to store several subproofs for the same subproblem, which is not supported by the current implementation of the $\mathcal{PDS}$.

6.3.1.2 Comparing MULTI with HEARSAY-III and BB1

Technically, MULTI is a simplified instantiation of the HEARSAY-III architecture. Conceptually, it comprises additional elements for goal-directed reasoning that are similar to capabilities of BB1. To compare MULTI with HEARSAY-III we shall point out similarities and differences of the architectures and the main cycles. We shall conduct the comparison of Multi and BB1 at the conceptual level by discussing whether and how Multi satisfies the behavioral goals for intelligent control-problem-solving stated in section 2.2.3 as a motivation for the design of BB1. We shall suggest possible extensions of Multi based on this comparison.

Multi **vs.** Hearsay-III

As Hearsay-III, Multi employs two different blackboards for the solution of the domain problem and the control problem. In Multi these blackboards are called the proof blackboard and the control blackboard. Moreover, as Hearsay-III, Multi distinguishes two kinds of knowledge sources working on these blackboards, namely strategies, which work on the proof blackboard, and the MetaReasoner, which is the only knowledge source working on the control blackboard. As the knowledge sources in Hearsay-III Multi's strategies are condition-action pairs. The activation records of Hearsay-III are called job offers in Multi and are maintained in a list on the control blackboard. Multi realizes a base scheduler as a loop that chooses the first job offer from this list and executes the corresponding strategy. Since there is only one knowledge source working on the control blackboard in Multi there is no need for several scheduling levels on the control blackboard as in Hearsay-III.

The main cycles of activation record/job offer creation, selection and execution are essentially the same in Multi and Hearsay-III. The only difference is that Multi's MetaReasoner is not triggered by particular events. Rather than placing job offers itself onto the control blackboard and competing with other knowledge sources, its execution is encoded into the control cycle of Multi (see Figure 6.2 on page 92). Another important difference between Multi and Hearsay-III is the duration of knowledge source executions. In Hearsay-III, knowledge source executions are indivisible: they run until completion and cannot be interrupted. In Multi, a strategy execution can be interrupted as described in section 6.2.2.

Multi **vs.** BB1

Multi satisfies the behavioral requirements that motivated the development of BB1 (see [111]) as follows:

- *Make explicit control decisions that solve the control problem.*
  This is realized in Multi by strategic control rules that explicitly reason on the job offers posed by the strategies.

- *Decide what actions to perform by reconciling independent decisions about what actions are desirable and what actions are feasible.*
  Multi satisfies this goal by explicitly distinguishing between the knowledge of when a strategy execution is feasible (stated in the condition of the strategy) and the knowledge of when a strategy execution is desirable (formalized in strategic control rules). Moreover, the reasoning processes on legal feasibility and heuristic desirability are strictly separated (see Multi's control cycle in Figure 6.2 on page 92).

- *Adopt, retain, and discard individual control heuristics in response to dynamic problem-solving situations.*
  Control heuristics are implemented in Multi's strategic control rules. In the current implementation it is not possible to change the set of strategic control rules during a run (see the following discussion of possible extensions of Multi).

- *Decide how to integrate multiple control heuristics of varying importance.*

In Multi it is possible to express a priority among the heuristics implemented in strategic control rules. However, in the current implementation of Multi there is no hierarchy notion for the employed heuristics as in the different levels of the BB1 control blackboard.

- *Dynamically plan strategic sequences of actions.*
  In the current implementation of Multi, it is not possible to plan whole sequences of strategy executions. However, posing demands and interrupting strategies allows for a goal-directed behavior in Multi that is a simple form of the goal-directed reasoning in BB1 (see the following discussion of possible extensions of Multi).

Several extensions of Multi could be considered in the future:

1. The goal-directed reasoning approach could be extended. For instance, there could be control knowledge sources that notice highly desirable strategies whose conditions are not satisfied. After analyzing the conditions of these strategies, such a control knowledge source would pose demands for other strategies whose executions likely enable the execution of a highly desirable strategy. A first example realizing such goal-directed reasoning in Multi is described in section 8.2.3 in the case studies. Here, a strategic control rule recognizes that a (desirable) strategy, which is supposed to be applicable, does not pose a job offer. As a consequence, the strategic control rule prefers a job offer whose execution will likely enable the desired strategy application.

2. Another approach to extend the goal-directed reasoning in Multi could be meta-planning at the strategy-level. Supposed the preconditions and the effects of the strategies are described in some formal language, then planning could be conducted at the strategy-level by special control knowledge sources. A plan of strategy executions and their relationships (e.g., which strategy execution is supposed to provide effects that another strategy execution requires as preconditions) could then influence the solution of the domain problem similar as demands. That is, strategic control rules analogous to `prefer-demand-satisfying-offers` could prefer job offers that correspond to steps in the strategy plan or — if there is no such job offer — they could prefer job offers that are likely to enable steps in the strategy plan.

3. BB1 allows to change the employed heuristics by placing control decisions onto the control blackboard. Similarly, it would be possible to place in Multi all control related issues on the control blackboard and to allow for their manipulation by particular knowledge sources. For instance, Multi could store all given strategies and strategic control rules on the control blackboard. The status of a strategy or a strategic control rule could be changed by knowledge sources from active to passive and vice versa. Multi would then consider only active strategies for invocation and the MetaReasoner would evaluate only active control rules.

The development of Multi and the introduction of the strategy-level for proof planning is due to the observation that there is a need for such a level. The evidence occurred in the experiments we conducted in the limit and the residue class domain. Although very interesting in general, it is not clear whether the possible extensions of Multi will be necessary and sensible for proof planning in the future. However, it is clear that all mentioned extensions would not only provide additional capabilities, but would also create further computational overhead. Hence, we did not include these features into the current implementation of Multi, but only suggest them as possible extensions, in case they are needed.

6.3.2 Knowledge Sources vs. Agents

MULTI employs a blackboard architecture in order to allow for the flexible cooperation of independent knowledge sources. However, there are also other AI-architectures for this purpose, in particular, multi-agent architectures. In this section, we shall discuss the question to what extend our knowledge sources qualify as agents and why we did not decide for a multi-agent system.

Currently, there is no universally accepted definition for the notion agent.[10] However, there is a consensus on at least some of the attributes a computational entity has to exhibit to be called an agent. In [248], WOOLDRIDGE identifies as essential property of an *agent* the capability of flexible, autonomous actions, which he characterizes with three abilities: *reactivity*, *pro-activeness*, and *social ability*.[11] Reactivity means that agents are robust in the sense that they can adapt to the changes in their environment. Pro-active means that agents exhibit not only goal-directed behavior but also take the initiative to pursue their goals. Finally, social abilities enable agents to negotiate with other agents to share goals and to cooperate.

In our architecture the strategies, that is, the knowledge sources of the proof blackboard, show some pro-active and some reactive characteristics. They are pro-active since they are not explicitly scheduled by a pre-defined control routine. Rather they become active themselves as soon as their condition part is satisfied. Then, they post job offers onto the control blackboard in order to indicate that they can contribute to the problem solving process. The strategies are partially reactive since they can adapt with respect to the information on the proof blackboard. For instance, since the control rules of strategies of **PPLANNER** rely on the proof context stored on the proof blackboard these strategies may introduce different actions in different proof contexts (for the same task).

The strategies lack social abilities. They can cooperate either in a data-driven manner in which a strategy becomes triggered by changes caused by another strategy or else on demand when one strategy explicitly interrupts and posts a demand for another strategy. There are no negotiations among the strategies in MULTI. Rather, the question which strategy to apply next is decided by the MetaReasoner, which evaluates the strategic control rules. If we distributed the heuristic knowledge encoded in the strategic control rules to all affected strategies, then the strategies could afterwards negotiate directly with each other which (applicable) one is the most desirable one. This would result in more autonomous entities, that comprise not only the knowledge of when they are applicable (knowledge of legal feasibility) but also of when it is useful that they are applied or when they should give precedence to other strategies (knowledge of heuristic utility).

Why did we decide for a separated encoding of the heuristic utility knowledge in control rules as opposed to the legal feasibility conditions of a strategy that are part of the strategy specification? The arguments for the separation at the strategy-level are essentially the same as at the method-level where the knowledge of the legal feasibility of the methods (in the application conditions of the methods) is separated from the knowledge of their heuristic utility (in control rules). First, knowledge becomes better manageable when developed and implemented in small, independent units. This also facilitates the knowledge acquisition process since it allows for a divide and conquer approach. Second, several experiments (e.g., see [176]) indicate the superiority of a separate representation of control knowledge in AI-

[10] NWANA and NDUMU characterize in [185] the current situation as follows: "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'.

[11] WOOLDRIDGE emphasizes that his definition of an intelligent agent is not accepted as a universally one.

planning. The separation facilitates modifications and learning since different kinds of knowledge can be modified independently, for instance, in order to experiment with different search controls or to learn new control at run-time.[12] Last but not least, mathematical problem solving favors the separation of control knowledge from other knowledge as Schoenfeld points out:

> *The perspective taken in this book is that it is useful to think of resources*[a] *and control as two qualitatively different, though deeply intertwined, aspects of mathematical behavior. This distinction raises delicate issues, for discussions of resources must include questions of access and attention that are, in a broad sense, issues of control.*
>
> Schoenfeld, 1985, [209] pp. 134–135
>
> [a] Schoenfeld mentions as resources of a particular domain: (1) informal and intuitive knowledge about the domain, (2) facts, definitions, and the like, (3) algorithmic procedures, (4) routine procedures, (5) relevant competencies, (6) knowledge about the rules of discourse in the domain (see [209] pp. 54–55).

6.3.3 Multi vs. ΩAnts

With Multi and ΩAnts (see section 3.2.4), Ωmega employs two blackboard-based components. A direct comparison of the two architectures (i.e., which elements of the one architecture relate to which elements in the other architecture) is not suitable since they serve different purposes. Rather, we shall point out the different purposes of ΩAnts and Multi and discuss how these objectives influenced their designs. In particular, we shall discuss how and why ΩAnts employs concurrency and why we do not perform similar processes in Multi concurrently.

The original motivation for ΩAnts was to support interactive proof construction with rules and tactics. Without ΩAnts, the user of Ωmega has to test the available tactics and rules, collectively called inference rules, in order to find an applicable one. In particular, finding suitable instantiations of the arguments and the parameters of an inference rule is a painstaking process. The ΩAnts mechanism frees the user from this work by providing the information about which inference rules are applicable in the actual proof situation. For each inference rule, ΩAnts employs a separate blackboard on which independent, concurrent knowledge sources, so-called agents, assemble information on possible applications of the inference rule. Applicable instantiations of the inference rule are reported by a monitoring agent to the suggestion blackboard. The entries of this blackboard are then provided as suggestions to the user who selects one.

For some inference rules, applicable instantiations can be found very quickly (if they exist); for other inference rules finding applicable instantiations can comprise time-consuming calls to external systems (e.g., ATPs) whose performance and result cannot be predicted. In order to avoid that the user has to wait for the next suggestions until all agents finish their computations ΩAnts employs the independent agents concurrently. This allows for an any-time behavior of the system, which immediately reports found instantiations to the user, who can then decide to apply one of the suggestions or to wait for further ones. Time consuming processes that are not finished, when the user selects a suggestion are not terminated but continue to run in the background.

Recent research aims to employ the ΩAnts mechanism also for automated proof construction. Instead of providing suggestions to the user a selector chooses and

[12] Although there are only preliminary approaches to learn search control in Ωmega so far (e.g., see section 9.2.2) we are planning to conduct further experiments on learning control knowledge.

applies a suggestion from the suggestion blackboard. The automated ΩANTS is envisaged for application in domains for which no or only little knowledge is available. In such domains, ΩANTS should perform proof search with rather general rules and tactics and with external systems. The idea is that the concurrent agents allow for the interleaving of repeated calls of external systems, in particular ATPs, with ongoing problem manipulation and (hopefully) simplification.

The control layer in ΩANTS is rather poorly developed so far. ΩANTS employs some general heuristics on which suggestions to pass from the rule blackboards to the suggestion blackboard as well as on how to rank the suggestions on the suggestion blackboard. The current selector simply takes the highest ranked suggestion. Although not developed to employ sophisticated control information like used in proof planning, the adaption of the control to different application domains will be necessary. However, it is not yet clear how further control information for domains can be used in ΩANTS. Another open research question is when to terminate resource-consuming processes.

In contrast to ΩANTS, MULTI's primary motivation was to develop a knowledge-based, automated component. MULTI can employ elaborate domain knowledge and sophisticated control knowledge. MULTI depends on this knowledge, such that it can be applied only to domains for which suitable knowledge has been acquired.

In section 6.3.1 we discussed already why the current implementation of MULTI does not enable the concurrent execution of several strategies. Another possibility to employ concurrency would be to evaluate strategic control rules while some strategies still check their condition parts. This would result in an any-time behavior like in ΩANTS. Although this would be technically possible, we decided for a sequential check of the condition parts and the subsequent evaluation of the strategic control rules since MULTI is a knowledge-based system in which an any-time behavior like in ΩANTS is not helpful.

If the MetaReasoner evaluated the control rules before all strategies posed their job offers onto the control blackboard, then its decisions would depend on which strategies did actually pose their job offers so far. Thereby, we would risk to miss the best strategy in the current situation since it did not pose a job offer so far. MULTI's philosophy is to acquire and formalize specific domain knowledge (which is a difficult work). If suitable domain knowledge is available it is not sensible to base the evaluation and incorporation of this knowledge on random effects such as which strategies did actually pose their job offers so far.[13] When the MetaReasoner waits until all strategies posed their job offers, then the concurrent check of the condition parts of the single strategies is only sensible when the checks are distributed to different processors. Since the condition parts of the strategies are rather simple functions so far, we did not consider a distribution, which would create much computational overhead.

6.4 Related Work

In the previous section we discussed aspects of MULTI's blackboard architecture and compared it with other blackboard architectures as well as with multi-agent architectures. In this section, we shall discuss peculiarities of proof planning with multiple strategies and compare it with related approaches from AI-planning and interactive and automated deduction.

[13] Note that for the concurrent computation and selection of actions in **PPLANNER** holds the same argument as for strategies: the decisions could depend on random effects, which is against the knowledge-based philosophy of ΩMEGA's proof planning.

We start with a comparison of the notion of a strategy in Multi with the notion usually used in AI-planning and automated deduction. Then, we compare the combination of strategies and algorithms possible in Multi with some approaches of AI-planning and automated deduction that combine different algorithms or different instances of an algorithm. Afterwards, we discuss how other proof planning systems use the notion strategy. We conclude with a discussion of the little theories approach realized in the Imps system and how it compares with the knowledge structuring realized in Multi.

6.4.1 Strategies in AI-Planning and Automated Deduction

In AI-planning as well as in machine-oriented automated deduction the notion of a strategy is typically used in the sense of a *search strategy*. A search strategy determines how the search space is traversed by influencing decisions at the choice points. For instance, an AI-planner following the precondition achievement paradigm has to decide which unsatisfied precondition to tackle next. If there are several actions that can satisfy the chosen precondition, it has also to decide which action to choose. A typical search strategy (or at least a part of a search strategy) in precondition achievement planning is to prefer that action that introduces the smallest number of new unsatisfied preconditions. A resolution-based ATP has to decide which clauses to use in the next resolution step. Common strategies for resolution-based ATPs assign weights to the clauses and then prefer clauses with the highest weights.

There is a wealth of work on search strategies that guide AI-planning systems and machine-oriented ATPs. Surveys on the subject are given in [33, 34] for automated deduction and in [194, 99] for AI-planners, where the interested reader will find extensive bibliographies.

Technically, search strategies in AI-planning and automated theorem proving as well as Multi's strategies all specify instances of parameterized algorithms. Proof planning with multiple strategies goes beyond the strategy concepts usually used in AI-planning and automated theorem proving by establishing facilities such as backtracking as separated algorithms in their own rights. Although **PPlanner** is Multi's main facility for the proof plan construction Multi is open for all kinds of refinement or modification algorithms that can contribute to the theorem proving process. The main difference between search strategies and **PPlanner** strategies is the kind of knowledge they comprise. Typically, a search strategy relies on domain-independent heuristics that hardly cover human proof or plan discovery heuristics. Since the heuristics are domain-independent their utility for a particular problem cannot be predicted. Thus, such a search strategy can perform totally different on similar problems of the same domain. **PPlanner** strategies, in contrast, comprise the knowledge of how to tackle a particular class of problems and try to integrate domain specific mathematical knowledge and practice. Moreover, Multi's strategies are condition-action pairs, that is, they explicitly comprise the knowledge to which class of problems they are applicable in their condition parts.

6.4.2 Combination of Systems and Strategies

Supposed there are several strategies for one system or several systems applicable to a problem, then the question is which strategy or which system should be applied to the problem. Contests among AI-planning systems[14] and machine-oriented

[14] See
ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html
http://www.cs.toronto.edu/aips2000/
http://www.dur.ac.uk/d.p.long/competition.html

ATPs[15], respectively, show that there is no system or strategy that outperforms all other systems or strategies in all domains. Hence, it is an obvious approach to combine different strategies or systems in order to extend the solvability horizon of the combined system. In the following, we shall discuss several approaches from AI-planning and machine-oriented automated theorem proving, which combine several strategies of one system (*homogeneous combination*) or several systems (*heterogeneous combination*). Another criterion to classify the approaches is whether they employ several strategies or systems in a *competitive manner* or in a *cooperative manner*. Several strategies or systems are applied in a competitive manner if each process obtains the complete problem as input and tries to find a solution for the problem where the processes are either time-sliced or parallelized. The combined system stops as soon as one process succeeds to prove the entire problem ("the winner takes it all"). Several strategies or systems work cooperatively if they can work on different subproblems of the overall problem and are able to exchange results. The combined system stops as soon as the integrated systems or strategies produce together a solution of the entire problem.

6.4.2.1 Combinations in AI-Planning

Fink describes in [87] the competitive selection of several strategies of the planner Prodigy. Prodigy provides several search strategies, which Fink calls "Search Engines". He uses the three search strategies APPLY, DELAY, and ALPINE. When choosing the strategy that should be applied to a problem, then there are two questions:

1. Which one is the most promising strategy for the problem at hand, that is, which should be tried first?
2. After which amount of time should the strategy be interrupted if it was not successful in order to try another strategy?

Fink's approach relies on a utility measurement for each strategy and a set of time bounds based on the experience about the performance of the three strategies on other problems. The strategy and the time bound with the highest estimated utility are chosen. It is not surprising that the three strategies solve in addition more problems than a single one. The remarkable result of the approach is that it was possible to compute suitable time bounds for the application of the strategies.

Whereas Fink uses several strategies of one planner, the competitive approach of Howe *et al.* relies on the choice among several planners [117]. Motivated from the results of the planner competition at AIPS 1998, which had no overall winner, Howe and colleagues used a meta-planner, called BUS, which can employ six planners: STAN, IPP, SGP, BlackBox, Ucpop, and Prodigy. For a given problem, BUS computes first for each system the estimated run time and the success probability. To estimate the run time and the success probability BUS examines certain features of the problem and its planning domain (e.g., the number of operators in the planning domain, or the number of goals of the problem). Then, it compares the features of the new problem and its planning domain with problems already tackled with the six planners. BUS orders the planners with respect to a certain average

for the results of the planner competitions held at the AIPS conferences 1998, 2000, 2002, respectively.

[15] See `http://www.cs.miami.edu/~tptp/CASC/17/`
`http://www.cs.miami.edu/~tptp/CASC/JC/`
for the results of the ATP competitions held at the CADE 2000 and the IJCAR 2001 conferences.

of predicted run time and predicted success probability and applies the systems sequentially in this order. First, each system is applied with its estimated run time as time bound. If one system succeeds, BUS terminates; otherwise it computes new time bounds and applies the planners again with these new time bounds until an overall time bound for the whole system is reached. Again, it is not surprising that six planners can solve more problems than a single one. But the experiments with BUS provide clear evidence that the average run time of the BUS system is considerably smaller than the average run times of the single planners — although the BUS system has an additional organization effort and the examined features for the performance analysis are rather general.

Wilkins and Myers propose in [245] the Multiagent Planning Architecture (MPA) as a framework for the cooperative integration of diverse technologies into a system capable of solving complex planning problems. Central in MPA is the notion of a *planning cell*. Planning cells are hierarchically organized collections of planning agents (PA) that are committed to one particular planning process. One cell employs different kinds of planning agents: Each cell has a meta-PA that serves as the manager of the cell, that is, it decomposes a planning task and distributes it to the PAs of the cell. Moreover, each cell employs a plan server, which provides the central repository for plans and plan-related information and makes this information accessible to all other cell agents. The plan server is a passive agent that responds to messages sent by other agents, but does not issue messages to other agents on its own initiative. Further PAs can employ existing software systems. In the application scenario in [245], PAs employ the Sipe-2 planner [246] and the Opis scheduler [220]. MPA allows for implementing several configurations of cells: a single cell configuration for generating individual solutions to a planning task, and a multiple-cell configuration for generating alternative solutions in parallel, where in multiple-cell configurations a meta-planning-cell manager distributes the problem to the single cells and collects their solutions.

6.4.2.2 Combinations in Automated Theorem Proving

There are several competitive approaches based on the SETHEO prover [145]. SETHEO is a theorem prover for first order predicate logic based on the model elimination calculus [146]. In [80] Ertel describes the RCTHEO system. RCTHEO employs a set of parallel processors, which all are running the same version of SETHEO in which the decisions at several choice points are randomized. Each copy of the randomized SETHEO is started with a different random seed. Since different random seeds produce different search paths they define different "strategies" of the randomized SETHEO. In [211] Schumann describes experiments with SiCoTHEO. As opposed to RCTHEO, in SiCoTHEO parallel processors run different instances of SETHEO that are created by varying certain pre-defined parameters that influence the traversal of the search space of SETHEO. In contrast to SETHEO, both systems, RCTHEO and SiCoTHEO, show super-linear speed-ups on certain problems. However, their success varies considerably among different problems. The idea of Wolf is that competing strategies should be complementary with respect to a given problem set, that is, the sets of problems solved in a certain time limit by two different strategies should differ "significantly". In [247] Wolf describes a methodology for computing schedules of complementary strategies with suitable time bounds based on experiments with training sets of problems. The approach is implemented in a system called p-SETHEO. Experiments with p-SETHEO evidence that the strategy schedules learned on a training set do outperform other strategy schedules on new problem sets.

Denzinger and Fuchs describe in [70] a methodology, the so-called TECHS approach (TEams for cooperative Heterogeneous Search), for achieving cooperation between several ATPs and several instances of them (i.e., several instances of one system have to use different search strategies). The experiments described in [70] use the systems Spass, SETHEO, and Discount. In the TECHS approach, (different) instances of the integrated systems form search teams. All included instances are wrapped with communication facilities that enable the interchange of selected intermediate results. This results in so-called search agents. The search of the single agents and the exchange of intermediate information is organized in cycles: during the working phase the single agents work independently and parallel on the given problem, whereas during the cooperation phase they exchange information. The interchanged information consists of clauses. Each agent employs so-called referees, which decide which clauses of the own search state should be communicated to the other agents and which clauses received from the other agents should be integrated into the search state. In the conducted experiments the TECHS approach clearly outperformed the single systems and their instances as well as a purely competitive parallel combination of them.

6.4.2.3 Comparison with Multi

Multi allows for both, the homogeneous combination of several strategies of one algorithm and the heterogeneous combination of different algorithms (via strategies of these algorithms). Moreover, Multi employs its strategies in a cooperative manner. With respect to these dimensions the TECHS and MPA approaches are the closest related ones to Multi. In the following, we shall compare some aspects of the three approaches.

Whereas TECHS prefers local, direct communication of partial results among the agents (i.e., the agents in TECHS communicate clauses), Multi and MPA use a central component in which the current solution state is stored: Multi stores the solution state in the elements of the blackboards, MPA uses a plan server. TECHS and MPA run their agents in parallel and on different machines whereas in Multi the strategies are scheduled sequentially and run on the same machine.

The three systems differ on what and how knowledge of the integrated components and their employment is represented and used. Multi emphasizes the formalization and incorporation of explicit knowledge of the applicability of strategies and the control of the search process. In its condition part each strategy comprises the knowledge on which tasks the strategy is feasible, and strategic control rules encode heuristic knowledge of the utility of strategy applications. In MPA, the knowledge of the employment of the agents is encoded into the manager of a planning cell. The manager distributes tasks to the single agents and assigns different responsibilities to them such as plan generation or scheduling. It is possible that the manager re-arranges the planning cell and changes the responsibilities of the agents. Hence, the responsibility of an agent is not part of the agent itself but is part of the manager of the planning cell, which stores it in a table. In TECHS, send-referees and receive-referees provide a possibility to encode knowledge of the combination of the agents by determining which clauses an agent communicates to other agents and which clauses it accepts from other agents. For instance, in the scenario described in [70] the provers Spass, SETHEO, and Discount were coupled. Since Discount is a pure equational prover only equational unit clauses are relevant for it. This knowledge can be encoded into the send-referees passing clauses to Discount or into the receive-referee accepting the clauses for Discount.

6.4.3 Notions of Strategies in Proof Planning

In the proof planners TIGER [184, 193] and λCLAM [204] there exist different notions of strategies, which we shall discuss in the following.

6.4.3.1 Structuring Incremental Proof Planning by Meta-Rule Sets

In the *incremental proof planning* approach [97] implemented in the TIGER system the central structure is a meta-rule. Meta-rules provide a declarative representation of the knowledge about the domain of application and about tactics. Technically, a meta-rule is a triple consisting of a precondition, an action, and a persistence condition (persistence conditions are optional). The preconditions and the persistence conditions are conjunctions of predicates on the current proof under construction. In the simplest case, an action is a tactic. In general, an action is a sequence of tactics and recursive calls to meta-rule sets interleaved with optional continuation conditions. Thus, meta-rules can be structured in meta-rule sets providing a further level of abstraction and structuring [98].

Proof planning with meta-rule sets works as follows: The planner is called with respect to a certain meta-rule set. First, the planner checks the preconditions of the given meta-rules and chooses one meta-rule whose precondition is satisfied. Then, the planner executes the action of the chosen meta-rule. If the action consists of one tactic, it applies this tactic. If the action consists of a sequence of tactics, it successively applies these tactics. If the application of one tactic in the sequence fails, the whole action fails and all tactics of the action already applied are retracted. If the action includes a call to another meta-rule set, the planner is invoked recursively with respect to this meta-rule set. If a meta-rule includes a persistence condition, the planner repeats the execution of the action of the meta-rule until the persistence condition is satisfied.

Meta-rule sets correspond to **PPLANNER** strategies in MULTI as a structuring mechanism for meta-rules or methods and control rules. Both approaches allow to interrupt a strategy/meta-rule set and to switch to another strategy/meta-rule set. MULTI goes beyond the capabilities of incremental proof planning with meta-rule sets by enabling the opportunistic, event-driven combination of strategies. This is possible since in its condition part a strategy includes an explicit representation of the knowledge to which tasks it is applicable. Moreover, control rules explicitly represent the heuristic knowledge about when the switch to another strategy is desirable. In contrast, in incremental proof planning each recursive invocation of a meta-rule set is encoded in some actions contained in other meta-rule sets. Neither the knowledge of the feasibility of a meta-rule set nor the knowledge of the desirability of a switch is explicitly represented. Thus, an opportunistic, event-driven combination of the meta-rule sets is not possible.

The flexible incorporation of algorithms for different proof plan refinements and modifications (e.g., backtracking, instantiation of variables, ATPs) is not covered by the strategies of incremental proof planning.

6.4.3.2 Compound Methods in λCLAM

Like in ΩMEGA also CLAM's and λCLAM's planning operators are called methods. A proof method in CLAM and λCLAM can be *atomic* or *compound*. A compound method is also called a strategy (e.g., see [69]).

Technically, strategies, i.e., complex methods, are constructed from simpler methods with constructors that are called *methodicals* [203] (in analogy to a tactical in LCF see section 3.2.2). For instance, (repeat_meth sym_eval) is a compound method that applies repeatedly the method sym_eval, which is itself again a compound method, while repeat_meth is a methodical. Other methodicals exist, for instance, for sequencing methods and creating OR choices, and, thus, complex proof strategies for controlling the search for a proof can be created successively. A proof strategy can also involve so-called *critics*, that is, procedures for reasoning on and patching of failures (see section 8.4 for a closer discussion of critics).

An example for a complex proof strategy realized in λCLAM is induction, which is implemented as a selection of atomic and compound methods. The top-level strategy induction_top_meth repeatedly attempts a disjunction of methods (i.e., methods connected with the OR methodical). These include basic tautology checking, generalization of common subterms and also symbolic evaluation and the induction strategy, ind_strat. Within ind_strat, the method induction_meth performs a ripple analysis to choose an induction scheme (from a selection specified in λCLAM's theories) and produces subgoals for base and step cases. The top-level strategy is applied once more to the base cases. The step cases are annotated and then the wave method is repeatedly applied to them followed by the method fertilize. Afterwards, the annotations are removed and the results are passed on to the top-level strategy again. The process terminates when all subgoals have been reduced to true.

Proof planning in λCLAM is similar to proof planning with meta-rule sets as discussed in the previous section. The user employs λCLAM with a compound method. Then, λCLAM processes the problem at hand with respect to the methodical expression of the compound method including recursive calls of other compound methods.

Proof planning in λCLAM does not separate heuristic control knowledge; rather, preconditions of methods may include legal and heuristic conditions. Thus, methods in λCLAM combine the functionalities of methods and control rules in ΩMEGA's proof planning. In particular, λCLAM uses *rippling*, a domain-independent difference reduction heuristic, which is encoded in the preconditions of the methods [43].

Similar to **PPLANNER** strategies in MULTI, compound methods provide a means to structure and restrict the available methods. Since compound methods have preconditions, the representation of knowledge of when the compound method is applicable and when a switch to the compound method is desirable would be possible. However, at present the preconditions of the compound methods are just true.[16] Switches among the compound methods are hard-coded into the compound methods and the methodicals they use and are not a choice point in its own right. Thus, an opportunistic, event-driven combination of compound methods like in MULTI is (currently) not possible.

As in incremental proof planning also in λCLAM the strategies do not cover the flexible incorporation of algorithms for different proof plan refinements and modifications such as backtracking, instantiation of variables, or ATPs).

6.4.4 Structuring Knowledge in Little Theories

In [82] FARMER and colleges present the *little theories* approach implemented in the IMPS system [81, 83] (Interactive Mathematical Proof System). The idea behind this approach is to employ a network of small axiomatic theories (i.e., theories that consist of small sets of axioms, respectively), called little theories, in order to develop a portion of mathematics with an interactive theorem proving system. Different theorems are proved in different theories, depending on the required knowledge.

[16] Personal communication with LOUISE DENNIS .

Apart from the fact that the use of fine-grained knowledge, the logical power of particular sets of axioms, and the relations among them are interesting research questions in their own rights, the little theories approach provides two practical benefits to the IMPS system:

1. It allows for minimal axiomatizations for specific groups of theorems.
2. It allows to make use of knowledge of the group of problems that should be tackled. In particular, so-called *processors* can be associated with a little theory. Processors are hand-coded algorithms that exploit facts about particular operators, either to simplify expressions or to decide formulas in some symbolic class. Processors may be far more efficient than the application of basic inferences to derive the same conclusion.

The first benefit facilitates the reuse of theorems in IMPS: The smaller the set of axioms on which a theorem depends the easier the theorem can be reused in other theories.[17] If the sets of axioms are very large, then the export of theorems into other theories becomes unmanageable. Similarly, strategies of **PPLANNER** allow to structure the methods and control rule knowledge. This is necessary in order to deal with the overwhelming knowledge that becomes unmanageable if not suitably structured (see section 6.1.4).

The second benefit reflects an insight that motivated and influenced the development of knowledge-based proof planning in general as well as MULTI's strategy approach in particular: mathematics of any complexity requires a mixture of different kinds of reasoning that have to be organized in order to be appropriately applicable. Similar to the processors in little theories, methods in ΩMEGA can perform steps particular to a certain domain or particular to a certain class of problems and a particular proof technique. Both little theories and strategies provide a means to organize the variety of available particular steps, simplifications, decision procedures and so on, such that the resulting units provide a means to tackle a certain class of problems.

6.5 Summary of the Chapter

In this chapter, we introduced the basic notions of proof planning with multiple strategies and its implementation in the MULTI system.

The development of of proof planning with multiple strategies was due to problems we encountered with ΩMEGA's previous planner PLAN. The conducted experiments for ϵ-δ-proofs and for residue class problems showed that PLAN's hard-coded integration of restricted components for action introduction, backtracking, and meta-variable instantiation represents one particular problem solving strategy suitable for many problems but insufficient as a general technique. Because of its rigid algorithm PLAN cannot be adapted to the needs of different problem classes and lacks any means to employ domain knowledge beyond methods and control rules, i.e., knowledge of different proof plan refinements and modifications and their

[17] Note that theories in ΩMEGA and IMPS are connected differently. The theories in IMPS form a network. Theories are connected by theory interpretations, which is a syntactic translation between two theories preserving theorems. That is, if a formula is a theorem of the source theory, then its image is a theorem of the target theory. When a theorem depends only of a minimal set of axioms, then this facilitates the export of the theorem to other theories and its reuse in these theories. The theories in ΩMEGA, in contrast, are arranged in a tree. An edge connects two theories $\mathcal{T}$ and $\mathcal{T}'$ when $\mathcal{T}'$ depends on $\mathcal{T}$, that is, $\mathcal{T}'$ inherits all axioms and definitions of $\mathcal{T}$. Thus, all theorems of $\mathcal{T}$ are automatically also theorems of $\mathcal{T}'$.

flexible combination. Our experiments illustrate that, in order to tackle a large body of problems, various proof plan refinements and modifications are necessary, and that the decision on when to call a certain refinement or modification should not be hard-coded into the system but rather be determined by meta-level reasoning using available heuristic control knowledge.

In order to meet these requirements, multiple-strategy proof planning decomposes the previous monolithic proof planning process and replaces it by separated parameterized algorithms for different kinds of plan refinements or modifications as well as different instances of these algorithms, which are called strategies. Heuristic control knowledge of the application and combination of the strategies can be encoded in strategic control rules.

To enable the flexible combination of strategies guided by the meta-level reasoning in the strategic control rules, we decided to implement MULTI in a blackboard architecture. Blackboard systems do not rely on a pre-defined control of the application of the involved components but provide the flexibility to employ their components, which are called knowledge sources, opportunistically. MULTI employs two separated blackboards: the proof blackboard contains the status and the history of the proof planning problem, the control blackboard contains the information relevant for the control problem, that is, which possible step should the system perform next. The strategies are the knowledge sources that work on the proof blackboard. An invoked strategy can refine or modify the proof plan under construction and records its changes in a history. The knowledge source that works on the control blackboard is called the MetaReasoner. It evaluates the strategic control rules in order to prefer or reject the application of strategies.

As compared with the previous proof planning, strategies and strategic control rules introduce another hierarchical level and its heuristic control. Moreover, they provide a means to encode and incorporate (mathematical) domain knowledge into the proof planning process beyond methods and method-level control rules. In the case studies in chapter 8, chapter 9, and chapter 10 we shall illustrate the available knowledge at the strategy-level and its importance for knowledge-based proof planning. However, before we discuss the case studies we first give a more technical description of the concepts in MULTI and the employed algorithms in the next chapter.

Chapter 7

Formal Description of Multi

In the previous chapter, we motivated and explained the design of Multi and its basic concepts. In this chapter, we shall give a formal description of Multi.

Proof planning with multiple strategies computes strategic actions and introduces them into a strategic proof plan. A strategic action is the instantiation of a strategy pattern corresponding to method actions, which are instantiations of methods. Similar to proof plans in PLAN a strategic proof plan consists of a sequence of actions, an agenda, and a $\mathcal{PDS}$. Strategic proof plans contain additionally a sequence of so-called binding stores to keep track of introduced meta-variable instantiations.

The structure of the chapter is as follows. First, we introduce some new data structures used by Multi among others binding stores. In section 7.2, we describe the different kinds of strategic actions in Multi. Afterwards, we formally describe strategic proof plans and give the operational semantics of strategic actions in section 7.3. Section 7.4 describes the strategic manipulation records, which Multi uses to construct a history. After the introduction of all necessary elements, we describe Multi's main cycle and the modification and refinement algorithms integrated so far in section 7.5. We conclude this chapter with the discussion of some particular technical features of Multi in section 7.6.

7.1 New Data Structures

In this section, we discuss some new data structures used in Multi and their role during the strategic proof planning process.

Binding Stores

Multi allows to reason on existing meta-variables and possible instantiations for them. An equation of the form $mv_\alpha :=^b t_\alpha$ where mv_α is a meta-variable and t_α is a term of the same type α is called a *binding*. t is called the *instantiation* of the binding for mv. During the strategic proof planning process the current set of bindings is stored in a so-called *binding store*.

New bindings are not applied to existing proof lines in the constructed $\mathcal{PDS}$ or to proof lines in existing actions. Since the application of the bindings would replace occurrences of the meta-variables by occurrences of their current instantiations, it would not be possible to backtrack binding decisions in order to bind meta-variables differently (since the information on which subterms of the proof lines have been which meta-variables would have been lost). Rather, the current bindings are applied to copies of proof lines as soon as these are used. For instance, if a line-task has the task formula $|mv_x - c| < c_\delta$ and the current binding store contains the binding $mv_x{:=}^b\, c$, then **PPlanner** applies the current binding to a copy of the task formula (see section 7.5.2 for details). The resulting formula, namely $|c - c| < c_\delta$, is then used in the action computation process instead of $|mv_x - c| < c_\delta$. Methods can become applicable wrt. the instantiated formula whereas they are not applicable wrt. the original formula with the meta-variables. For our example, a method for arithmetic simplifications becomes applicable and can reduce the formula $|c-c| < c_\delta$ to $0 < c_\delta$ which is not possible for $|mv_x - c| < c_\delta$. However, this step depends on the binding of mv_x; if this binding is removed (by backtracking the step that introduced the binding), then this step is not valid anymore.

Multi constructs a sequence of binding stores in order to keep track of the dependencies between the changing bindings and the introduced actions. The introduction of a new binding creates a new binding store in the sequence. All following steps are performed with respect to this current binding store. When bindings are removed, then the binding store before the introduction of this binding is restored and all following binding stores are removed from the sequence. Moreover, all actions that potentially depend on the removed binding stores are deleted as well (for details see section 7.5.7 where backtracking in Multi is described). We extended the notion of an action in proof planning for Multi. Actions have an additional slot **binding-store** in order to store a pointer to the binding store that was the current one when the action was computed.

Notation 7.1: In the remainder of the thesis, the following symbols (maybe labeled with some subscripts or superscripts) are associated with the following objects:
$\mathcal{B}$S denotes a binding store,
$\vec{\mathcal{B}}$S denotes a sequence of binding stores.

Task Tags

In Multi, a strategy is executed with respect to a particular task (from the blackboard point of view we can say that the existence of the task triggers the invocation of the strategy). A particular execution of a strategy tackles then the task by which it was triggered rather than arbitrary tasks. This is easy to realize for the algorithms **Exp**, **ATP**, and **InstMeta** since these algorithms perform just one refinement step before they terminate. The situation is more complicated for the algorithms **PPlanner** and **CPlanner** since they may perform a sequence of proof plan modifications (e.g., introduce several actions) before they terminate or interrupt. When applied with respect to an initial task, these algorithm should tackle this task and tasks that are derived from it but they should ignore other tasks in the agenda. Moreover, if a strategy execution of **CPlanner** or **PPlanner** interrupts and other strategies are executed, then some of these strategies work on tasks created by the interrupted strategy some of them work on other tasks. When the initial strategy is re-invoked again, then it should tackle tasks derived from its own tasks but it should ignore other tasks created meanwhile. To organize this behavior a maintenance mechanism is needed, which keeps track of which tasks are relevant for which strategies.

In Multi, the desired behavior is supported by so-called *task tags*. When a strategy of **CPlanner** and **PPlanner** is invoked, then it creates a new task tag $@_T$, which uniquely refers to this execution of the strategy. The task tag is pinned to the task that triggered the strategy. When a proof plan modification in Multi reduces a task to some new tasks, then the new tasks inherit all tags from the initial one. An execution of a strategy of **CPlanner** or **PPlanner** considers only tasks that carry its tag. When the strategy execution terminates, then its tag is removed from all tasks. When a strategy execution interrupts and is re-invoked later on, then the re-invocation continues to work with the task tag created by the initial invocation.

If used in several not-terminated strategies, then one task can carry several tags. For instance, when an execution of a **PPlanner** strategy creates a task T, then T carries the tag of this execution. Afterwards, the execution interrupts and a different strategy is applied to T. Then, this second strategy execution creates a new tag, which is also pinned to T. All actions introduced by this second strategy execution inherit both tags of T. When the second strategy execution terminates and its tag is removed, then the resulting tasks carry still the tag of the first strategy execution. Thus, when the first strategy execution is re-invoked, it can continue to tackle these tasks.

Note that the task tags describe only which tasks can be tackled by a strategy execution. This does not mean that the other tasks are "invisible" or temporarily removed. Control rules evaluated by **CPlanner** and **PPlanner** can reason on all tasks of the current agenda.

Execution Messages

When a strategy execution stops, then its result and the reason why it stops are relevant information for Multi since Multi treats different kinds of termination differently (see section 7.5). Moreover, this information is important for the meta-reasoning with strategic control rules. Therefore, each strategy execution in Multi stops with a so-called *execution message*, which contains the available termination information. So far, Multi uses the following execution messages:

- A *success message* occurs when the strategy execution is successful on the given task.
- A *failure message* occurs when the strategy execution fails on the given task because of some problems (e.g., a strategy of **PPlanner** fails because there are no further applicable actions).
- An *interruption message* occurs when a strategy of **CPlanner** or **PPlanner** is interrupted.

The algorithms can attach further information to the execution messages, which can also be used by the strategic control rules. For instance, an algorithm can attach information on what kind of failure occurred to a failure message (see section 7.6.5).

Execution messages are stored in the history entries created by the strategy executions (see section 7.4). When which algorithm terminates with which execution message is described in detail in section 7.5. When a strategy execution terminates with a success message we also say that *the application of the strategy was successful*.

Demands and Memory Entries

For the algorithms **CPlanner** and **PPlanner** a strategy execution can interrupt. If this is the case, the strategy execution creates so-called demands and adds them to the demand repository on the control blackboard. Multi knows for the following *demands*:

- A demand $S - ON - T$, which specifies a strategy S and a task T, is called a *strategy-task-demand*. This demand is satisfied by a successful application of the strategy S to the task T.

- A demand $S - ON-?$, which specifies a strategy S but no task, is called a *strategy-demand*. This demand is satisfied by a successful application of the strategy S to any task.

- A demand $? - ON - T$, which specifies a task T but no strategy, is called a *task-demand*. This demand is satisfied by a successful application of any strategy to the task T.

An interrupted strategy execution writes also an entry into the memory repository on the control blackboard. A *memory entry* is a pair $(@_T, \{P_{D_1}, \dots, P_{D_n}\})$ of a task tag $@_T$ and a set of pointers $\{P_{D_1}, \dots, P_{D_n}\}$ to the demands of the interrupted strategy execution in the demands repository. MULTI uses the $@_T$ to re-invoke the strategy execution later on (see section 7.5.2 for details). Moreover, it makes use of the pointers to check whether the demands of the interrupted strategy are satisfied such that the strategy execution can be re-invoked again (see section 7.5.1 for details).

7.2 Strategic Actions

PLAN computes and introduces actions into a proof plan. An action is an instantiation of a method, which is a pattern of a proof step (see section 4.1.2). To extend this approach of action computation and introduction to strategic proof planning there is a strategic pattern associated with each algorithm in MULTI (except **BACKTRACK**). The application of a strategy computes an instantiation of the pattern of its algorithm, a so-called *strategic action*, and introduces it into the strategic proof plan.

In this section we shall describe the strategic actions created by the algorithms **PPLANNER**, **INSTMETA**, **EXP**, **ATP**, and **CPLANNER**. The algorithm **BACKTRACK** does not create actions but deletes actions of other algorithms. Note that, henceforth, we call instantiations of methods *method actions* in order to distinguish them from the different strategic actions, which we call **PPLANNER** *actions*, **INSTMETA** *actions*, **EXP** *actions*, **ATP** *actions*, and **CPLANNER** *actions*.

Technically, strategic actions are implemented as frame data structures. Each strategic action has the slots **strategy**, **task**, and **binding-store**. The *strategy of an action* and the *task of an action* are pointers to the strategy and the task with respect to which the action was computed. The *binding store of an action* is a pointer to the binding store, which was the current binding store, when the action was computed. Depending on the algorithm the different strategic actions have also further slots.

PPLANNER and CPLANNER

The algorithms **PPLANNER** and **CPLANNER** successively introduce actions into a strategic proof plan, **PPLANNER** with respect to a given set of methods and control rules, **CPLANNER** with respect to a given plan or a given plan fragment. Thus, actions of **PPLANNER** and **CPLANNER** are essentially abstractions of the sequence of actions introduced by the respective algorithm. The sequence of introduced actions is stored in the slot **action-sequence** of a **PPLANNER** or **CPLANNER** action.

Executions of **PPlanner** and **CPlanner** strategies can interrupt and can be reinvoked later on. Thus, one execution can consist of several periods. **PPlanner** and **CPlanner** create a strategic action for each period of the same strategy execution. Each of these actions contains the initial task to which the strategy was applied in the **task** slot. In its **action-sequence** slot each action contains only those actions that were introduced during the corresponding execution period. Note that the information stored in the strategic actions is not sufficient to identify actions that belong to the same strategy execution. For that purpose also information stored in the corresponding history entries is needed (see section 7.4 for details on the history entries).

PPlanner Action	
strategy	NormalizeLineTask
task	$L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash \exists x_\bullet(0 < x \wedge F[x])\ (open) \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$
binding store	$\mathcal{BS}$
action-sequence	$[A_{\exists\text{I-B}}, A_{\wedge\text{I-B}}, \ldots]$

Figure 7.1: A strategic action of **PPlanner**.

An example for an action of **PPlanner** is given in Figure 7.1. The strategic action results from the application of the strategy NormalizeLineTask to the line-task $L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash \exists x_\bullet(0 < x \wedge F[x])\ (open) \blacktriangleleft \{L_{Ass_1}, L_{Ass_2}\}$. First, **PPlanner** applies the method ∃I-B to the initial task. Then, it applies the method ∧I-B to the resulting task with task-formula $0 < mv_x \wedge F[mv_x]$. If $F[mv_x]$ is again a complex formula, then **PPlanner** can perform further actions in order to decompose $F[mv_x]$. The sequence of actions performed by **PPlanner**, $[A_{\exists\text{I-B}}, A_{\wedge\text{I-B}}, \ldots]$, is stored in the slot **action-sequence** of the strategic action.

ATP

The algorithm **ATP** employs external automated theorem provers to prove line-tasks. If the automated theorem prover succeeds, then the **ATP** algorithm closes the goal of the line-task and creates a strategic action and stores the output of the external system in the slot **output**.

An example for an action of **ATP** is given in Figure 7.2. The strategy CallTramp is applied to the (trivial) problem to show that $P \Rightarrow P$ holds. The problem is passed to Tramp, which provides as output the ND-proof given in the **output** slot of the action.

ATP Action	
strategy	CallTramp
task	$L.\emptyset \vdash\ P \Rightarrow P\ (open) \blacktriangleleft \emptyset$
binding store	$\mathcal{BS}$
output	$L_1.\quad L_1\quad \vdash P\qquad (Hyp)$ $L_2.\quad L_1\quad \vdash P\qquad (Weaken)$ $L.\quad \emptyset\quad \vdash P \Rightarrow P\qquad (\Rightarrow_I\ L_2)$

Figure 7.2: A strategic action of **ATP**.

EXP

The algorithm **EXP** expands complex steps, i.e., method or tactic steps in the constructed $\mathcal{PDS}$. For a proof line L with justification $(J\ P_1\ \dots P_n)$, where J is a method or a tactic and $P_1, \dots, P_n$ are the premises, **EXP** computes a proof segment that derives the conclusion L of the step from its premises $P_1, \dots, P_n$ at a lower level of abstraction. This proof segment is stored in the slot **expansion-segment** of an action of **EXP**. Moreover, an **EXP** action contains the slot **open-lines**, which contains the set of new open lines that are introduced in the expansion-segment.[1]

An example is given in Figure 7.3. This **EXP** action results from the expansion of the justification (=Subst-B $L_{Thm'}$ L_{Ass_1}) of proof line L_{Thm} (compare with example 4.5 in section 4.1.2). When this step is expanded, then the proof schema of the method =Subst-B (see section 4.1.1) is instantiated in order to derive L_{Thm} from the premises $L_{Thm'}$ and L_{Ass_1} as given in the **expansion-segment** in Figure 7.3.

EXP Action	
strategy	**EXP**
task	$L_{Thm}.\ L_{Ass_1}, L_{Ass_2}\ \vdash even(a+b)$ (=Subst-B $L_{Thm'}\ L_{Ass_1})\vert^{Exp}$
binding store	$\mathcal{B}$S
expansion-segment	$L_{Ass_1}.\quad L_{Ass_1}\quad \vdash a \doteq c\quad (Hyp)$ $L_{Thm'}.\quad L_{Ass_1}, L_{Ass_2} \vdash even(c+b)\quad (Open)$ $L_1.\quad L_{Ass_1}, L_{Ass_2} \vdash \forall P\mathbf{.}P(c) \Rightarrow P(a)\quad (\equiv_E\ L_{Ass_1}\ (\doteq))$ $L_2.\quad L_{Ass_1}, L_{Ass_2} \vdash (\lambda x\mathbf{.}even(x+b))(c) \Rightarrow (\lambda x\mathbf{.}even(x+b))(a)\quad (\forall_E\ L_1\ (\lambda x\mathbf{.}even(x+b)))$ $L_3.\quad L_{Ass_1}, L_{Ass_2} \vdash even(c+b) \Rightarrow even(a+b)\quad (\lambda\!\leftrightarrow\ L_2)$ $L_{Thm}.\quad L_{Ass_1}, L_{Ass_2} \vdash even(a+b)\quad (\Rightarrow_E\ \ L_3\ L_{Thm'})$
open-lines	$\{\}$

Figure 7.3: A strategic action of **EXP**.

INSTMETA

The algorithm **INSTMETA** computes instantiations of meta-variables. An action of **INSTMETA** stores the computed instantiation in the slot **instantiation**. An example for an action of **INSTMETA** is given in Figure 7.4. This action results from the application of the strategy ComputeInstFromCS to the task $mv_\delta\vert^{Inst}$. **INSTMETA** computes the instantiation $min(c_{\delta_1}, c_{\delta_2})$ for mv_δ and stores it in the **instantiation** slot.

INSTMETA Action	
strategy	ComputeInstFromCS
task	$mv_\delta\vert^{Inst}$
binding store	$\mathcal{B}$S
instantiation	$min(c_{\delta_1}, c_{\delta_2})$

Figure 7.4: A strategic action of **INSTMETA**.

[1] If one of the premises $P_1, \dots, P_n$ is open, then it is not in this slot, since it was not changed by the expansion (i.e., its open justification was not created by the expansion).

7.3 Strategic Proof Plans

In this section, we shall extend the notions introduced in section 4.2.1 to strategic proof plans. We start with the definitions of a strategic proof planning problem, an initial $\mathcal{PDS}$ of a strategic proof planning problem (which is the same as the initial $\mathcal{PDS}$ of a proof planning problem), and an initial agenda of a strategic proof planning problem (which is different from the initial agenda of a proof planning problem since it may contains instantiation-tasks).

Definition 7.2 (Strategic Proof Planning Problem):
A *strategic proof planning problem* is a quadruple $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$, where Thm and $Ass_1, \ldots, Ass_n$ are formulas in ΩMEGA's higher-order language, $\mathcal{S}$ is a set of strategies, and $\mathcal{C}_{\mathcal{S}}$ is a set of strategic control rules. Thm is also called the *theorem* of the strategic proof planning problem whereas $Ass_1, \ldots, Ass_n$ are called the *assumptions* of the strategic proof planning problem. □

Definition 7.3 (Initial $\mathcal{PDS}$, Initial Agenda):
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$ be a strategic proof planning problem. The *initial $\mathcal{PDS}$* of this problem is the $\mathcal{PDS}$ that consists of an open line L_{Thm} with formula Thm and the lines L_{Ass_i} with formula Ass_i and the hypothesis justification Hyp, respectively. The *initial agenda* of the strategic proof planning problem is the agenda that consists of the line-task $L_{Thm} \blacktriangleleft \{L_{Ass_1}, \ldots, L_{Ass_n}\}$ and an instantiation-task $mv|^{Inst}$ for each meta-variable in $L_{Thm}, L_{Ass_1}, \ldots, L_{Ass_n}$. □

Next, we extend the action applicability notion of PLAN. In MULTI, actions are applicable with respect to a $\mathcal{PDS}$ and a binding store. In particular, an action is applicable only if the current binding store equals[2] the binding store with respect to which the action was computed (i.e., the binding store that is stored in the slot **binding store** of the action). This restriction is necessary since the computation of actions can rely on given bindings in the current binding store. Moreover, we extend the action introduction functions Φ and $\vec{\Phi}$ of PLAN (see definition 4.11 and definition 4.12) to the strategic action introduction functions $\Phi_{\textsc{Multi}}$ and $\vec{\Phi}_{\textsc{Multi}}$. $\Phi_{\textsc{Multi}}$ describes the operational semantics of an action in MULTI when it is applied to an agenda, a $\mathcal{PDS}$, a sequence of actions, and a sequence of bindings stores, i.e., $\Phi_{\textsc{Multi}}$ defines a transition relation between quadruples of agendas, $\mathcal{PDS}$s, sequences of actions, and sequences of binding stores. First, we give general definitions of $\Phi_{\textsc{Multi}}$ and $\vec{\Phi}_{\textsc{Multi}}$. Then, we define for each kind of action used in MULTI when it is applicable and the results of its introduction by $\Phi_{\textsc{Multi}}$.

Definition 7.4 (Action Introduction Functions $\Phi_{\textsc{Multi}}$ and $\vec{\Phi}_{\textsc{Multi}}$): The *action introduction function* $\Phi_{\textsc{Multi}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and an applicable action into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\Phi_{\textsc{Multi}} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\mathrm{S} \times A_{add} \mapsto \vec{A}' \times \hat{\mathrm{A}}' \times \mathcal{P}' \times \vec{\mathcal{B}}\mathrm{S}'.$$

The *recursive action introduction function* $\vec{\Phi}_{\textsc{Multi}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and a sequence of actions into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\vec{\Phi}_{\textsc{Multi}} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\mathrm{S} \times \vec{A}_{add} \mapsto \vec{A}' \times \hat{\mathrm{A}}' \times \mathcal{P}' \times \vec{\mathcal{B}}\mathrm{S}'.$$

[2] Two binding stores are equal when they contain the same bindings.

$\vec{\Phi}_{\text{MULTI}}$ is recursively defined as follows:
Let $\vec{A}$ be a sequence of actions, Â an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, $\vec{\mathcal{B}}$S a sequence of binding stores, and $\vec{A}_{add}$ a sequence of actions.

1. If $\vec{A}_{add}$ is empty, then
$\vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{add}) := (\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S})$.

2. Otherwise let $A_{add} :=$ *first*$(\vec{A}_{add})$ and $\vec{A}\text{'}_{add} :=$ *rest*$(\vec{A}_{add})$. If A_{add} is applicable with respect to $\mathcal{P}$ and the last binding store of $\vec{\mathcal{B}}$S, and if Â contains the task of A_{add}, then
$\vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, \vec{A}_{add}) := \vec{\Phi}_{\text{MULTI}}(\Phi_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{add}), \vec{A}\text{'}_{add})$.

□

Method Actions

A method action is applicable with respect to a $\mathcal{PDS}$, if the given lines of the action are in the $\mathcal{PDS}$. Φ_{MULTI} differs from Φ in two points. First, Φ_{MULTI} creates not only new line-tasks but also new instantiation-tasks (for each new meta-variable in the new outlines created by the method action) and new expansion-tasks (for each conclusion of the method action). Second, MULTI allows method actions that contain binding constraints in their **constraints** slot. These binding constraints are labeled with *Binding*, which indicates that they are not passed to an external constraint solver but to the binding store.[3] When the action is introduced, a new binding store is created and added to the sequence of binding stores. The new binding store results from the union of the bindings of the last binding store and the new bindings. The instantiation-tasks whose meta-variables are bound by the new bindings are then removed from the agenda.

Definition 7.5 (Applicable Method Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$, $\mathcal{B}$S a binding store, and A_{add} a method action with the binding store $\mathcal{B}\text{S}_{A_{add}}$. Moreover, let $\mathcal{L}$ be the set of proof lines of $\mathcal{P}$ and let $\ominus Concs$ be the $\ominus$ conclusions, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. $(\ominus Concs \cup \ominus Prems \cup BPrems)$ is a subset of $\mathcal{L}$,

2. $\mathcal{B}\text{S}_{A_{add}} = \mathcal{B}\text{S}$.

□

Definition 7.6 (Φ_{MULTI} on Method Actions): Let $\vec{\mathcal{B}}$S be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}$S. Let $\vec{A}$ be a sequence of actions and let A_{add} be a method action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $\oplus Concs$ be the $\oplus$ conclusions, $\ominus Concs$ the $\ominus$ conclusions, $\oplus Prems$ the $\oplus$ premises, $\ominus Prems$ the $\ominus$ premises, and $BPrems$ the blank premises of A_{add}. Let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add} and let σ be the binding constraints of A_{add}.
$Prems$:=$\oplus Prems \cup \ominus Prems \cup BPrems$,
$Concs$:=$\oplus Concs \cup \ominus Concs$

[3] Internal binding constraints in method actions were first introduced by LASSAAD CHEIKHROUHOU in an extension of PLAN for proof planning diagonalization proofs [49].

$\mathit{New\text{-}Lines}{:=}\oplus \mathit{Concs} \cup \oplus \mathit{Prems}$
$\mathit{New\text{-}Supps}{:=}(\mathcal{SUPPS}_{L_{open}} \cup \oplus \mathit{Concs}) - \ominus \mathit{Prems}.$
$\mathit{New\text{-}Line\text{-}Tasks}{:=}[L \blacktriangleleft \mathit{New\text{-}Supps} \mid L \in \oplus \mathit{Prems}].$
$\mathit{New\text{-}Inst\text{-}Tasks}{:=}[mv|^{Inst} \mid mv \in \mathit{New\text{-}Lines}$ *and not* $mv|^{Inst}$ *in* $\hat{\mathrm{A}}].$
$\mathit{New\text{-}Exp\text{-}Tasks}{:=}[C|^{Exp} \mid C$ *in* $\mathit{Concs}].$
$\mathit{New\text{-}Tasks}{:=}\mathit{New\text{-}Line\text{-}Tasks} \cup \mathit{New\text{-}Inst\text{-}Tasks} \cup \mathit{New\text{-}Exp\text{-}Tasks}.$
$\mathit{Old\text{-}Inst\text{-}Tasks}{:=}[mv|^{Inst} \mid mv{:=}^b t \in \sigma].$
$\hat{\mathrm{A}}_{rest}{:=}\hat{\mathrm{A}} - ([\mathrm{T}] \cup \mathit{Old\text{-}Inst\text{-}Tasks})).$

If $\hat{\mathrm{A}}$ is an agenda that contains the task T of A_{add}, then the result $(\vec{A}', \hat{\mathrm{A}}', \mathcal{P}', \vec{\mathcal{B}}\mathrm{S}')$ of $\Phi_{\mathrm{MULTI}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, A_{add})$ is defined by:

- $\vec{A}'{:=}\ \vec{A} \cup [A_{add}].$
- $\hat{\mathrm{A}}'{:=} \begin{cases} \mathit{New\text{-}Tasks} \cup \hat{\mathrm{A}}_{rest} & \text{if } L_{open} \in \ominus \mathit{Concs}, \\ [L_{open} \blacktriangleleft \mathit{New\text{-}Supps}] \cup \mathit{New\text{-}Tasks} \cup \hat{\mathrm{A}}_{rest} & \text{else.} \end{cases}$
- $\mathcal{P}'$ results from $\mathcal{P}$ by
  1. adding the proof lines *New-Lines*, respectively, and
  2. justifying the proof lines $\ominus$*Concs* and $\oplus$*Concs* by the application of the method of A_{add} to *Prems*, respectively.
- If σ is empty, then $\vec{\mathcal{B}}\mathrm{S}'{:=}\vec{\mathcal{B}}\mathrm{S}$. Otherwise, $\vec{\mathcal{B}}\mathrm{S}'{:=}\vec{\mathcal{B}}\mathrm{S} \cup [\mathcal{B}\mathrm{S}_{new}]$ where $\mathcal{B}\mathrm{S}_{new} := \{mv_i{:=}^b\, t_i\sigma | (mv_i{:=}^b\, t_i) \in \mathcal{B}\mathrm{S}\} \cup \sigma$.[4]

□

INSTMETA Actions

An **INSTMETA** action is applicable with respect to a binding store and a $\mathcal{PDS}$, if the proof lines of the $\mathcal{PDS}$ contain occurrences of its meta-variable but there is no binding for the meta-variable in the binding store. When applied to an action of **INSTMETA**, Φ_{MULTI} creates a new binding store, which is added to the sequence of binding stores. The new binding store results from adding a binding for the meta-variable of the instantiation-task of the action to the last binding store of the sequence.

Definition 7.7 (Applicable INSTMETA Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$ with proof lines $\mathcal{L}$, $\mathcal{B}\mathrm{S}$ a binding store, and A_{add} an **INSTMETA** action. Let $T_{A_{add}} = mv|^{Inst}$ be the task of A_{add} and $\mathcal{B}\mathrm{S}_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}\mathrm{S}$, if

1. there are occurrences of mv in the formulas of the proof lines $\mathcal{L}$,
2. there is no binding for mv in $\mathcal{B}\mathrm{S}$,
3. $\mathcal{B}\mathrm{S}_{A_{add}} = \mathcal{B}\mathrm{S}$.

□

Definition 7.8 (Φ_{MULTI} on INSTMETA Actions): Let $\vec{\mathcal{B}}\mathrm{S}$ be a sequence of bindings stores and let $\mathcal{B}\mathrm{S}$ be the last binding store of $\vec{\mathcal{B}}\mathrm{S}$. Let $\vec{A}$ be a sequence of actions

[4] $t_i\sigma$ is the term that results from the application of the binding constraints in σ to the subterms of t_i. That is, each occurence of a meta-variable mv' in t_i that is bound by a constraint $mv{:=}^b\, t'$ in σ is replaced by an occurence of t'.

and let A_{add} be an **InstMeta** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = mv|^{Inst}$ be the task of A_{add} and let t be the instantiation for mv in A_{add}.
σ:=$\{mv{:=}^b t\}$.

If Â is an agenda that contains the task T of A_{add}, then the result ($\vec{A}$', Â', $\mathcal{P}$', $\vec{\mathcal{B}}$S') of $\Phi(\vec{A}$, Â, $\mathcal{P}, \vec{\mathcal{B}}S, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= Â - $[T]$.
- $\mathcal{P}$':= $\mathcal{P}$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S $\cup$ [$\mathcal{B}$S$_{new}$] where $\mathcal{B}$S$_{new}$:= $\{mv_i{:=}^b t_i\sigma | (mv_i{:=}^b t_i) \in \mathcal{B}S\} \cup \sigma$.

□

ATP Actions

An **ATP** action is applicable with respect to a $\mathcal{PDS}$, if the proof lines of the line-task of the action are in the $\mathcal{PDS}$. When applied to an action of **ATP** with task $L_{open} \blacktriangleleft \{S_1, \ldots, S_n\}$, Φ_{MULTI} closes L_{open} in the $\mathcal{PDS}$ with an application of the tactic *atp*. The only resulting new task is an expansion-task for L_{open}.

Definition 7.9 (Applicable ATP Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$ with the proof lines $\mathcal{L}$, $\mathcal{B}$S a binding store, and A_{add} an **ATP** action. Let $T_{A_{add}} = L_{open} \blacktriangleleft \{S_1, \ldots, S_n\}$ be the task of A_{add} and $\mathcal{B}$S$_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. $L_{open} \in \mathcal{L}$ and $\mathcal{SUPPS}_{L_{open}} \subseteq \mathcal{L}$,
2. $\mathcal{B}$S$_{A_{add}}$ = $\mathcal{B}$S.

□

Definition 7.10 (Φ_{MULTI} on ATP Actions): Let $\vec{\mathcal{B}}$S be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}$S. Let $\vec{A}$ be a sequence of actions and let A_{add} be an **ATP** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$ be the task of A_{add} and let Out be the content of the slot **output** of A_{add}.

If Â is an agenda that contains the task T of A_{add}, then the result ($\vec{A}$', Â', $\mathcal{P}$', $\vec{\mathcal{B}}$S') of $\Phi(\vec{A}$, Â, $\mathcal{P}, \vec{\mathcal{B}}S, A_{add})$ is defined by:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= (Â $- [T]) \cup [L_{open}|^{Exp}]$.
- $\mathcal{P}$' results from $\mathcal{P}$ by justifying the proof line L_{open} with an application of the tactic *atp* to the supports $\mathcal{SUPPS}_{L_{open}}$ and the parameter Out.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

EXP Actions

An **EXP** action is applicable with respect to a $\mathcal{PDS}$, if the closed line in the expansion-task of the action is in the $\mathcal{PDS}$ and if the premises of the justification of the closed line are in the $\mathcal{PDS}$. When applied to an action of **EXP**, Φ_{MULTI} introduces the new proof lines of the **expansion-segment** slot into the $\mathcal{PDS}$ and adds all resulting new tasks to the agenda, namely new instantiation-tasks for new meta-variables in the new proof lines, new line-tasks for open lines in the new proof lines, and new expansion-tasks for all new proof lines, which have a tactic or a method justification.

Definition 7.11 (Applicable EXP Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$ with the proof lines $\mathcal{L}$, $\mathcal{B}$S a binding store, and A_{add} an **EXP** action with the binding store $\mathcal{B}\text{S}_{A_{add}}$. Moreover, let $T_{A_{add}} = L|^{Exp}$ be the task of A_{add} where L has the justification $(J\ P_1\ \ldots P_n)$. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if

1. $L \in \mathcal{L}$ and $\{P_1\ \ldots P_n\} \subseteq \mathcal{L}$,
2. $\mathcal{B}\text{S}_{A_{add}} = \mathcal{B}\text{S}$.

□

Definition 7.12 (Φ_{MULTI} on EXP Actions): Let $\vec{\mathcal{B}}$S be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}}$S. Let $\vec{A}$ be a sequence of actions and let A_{add} be an **EXP** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $T = L|^{Exp}$ be the task of A_{add} and $(J\ P_1 \ldots P_n)$ the justification of L (before the expansion).
$\mathcal{SUPPS}$:=$\{P_1, \ldots, P_n\}$.
New-Lines:=**expansion-segment** of A_{add} without $L, P_1, \ldots, P_n$.
New-Open-Lines:=**open-lines** of A_{add}.
New-Line-Tasks:=[L' ◀ $\mathcal{SUPPS}$ | L' *in New-Open-Lines*].
New-Inst-Tasks:=[$mv|^{Inst}$ | $mv \in$ *New-Lines and not* $mv|^{Inst}$ *in* Â].
New-Exp-Tasks:=[$L'|^{Exp}$ |($L' \in$ *New-Lines or* $L' = L$) *and*
L' closed by tactic or method]
New-Tasks:=*New-Line-Tasks* ∪ *New-Inst-Tasks* ∪ *New-Exp-Tasks*.

If Â is an agenda that contains the task T of A_{add}, then the result ($\vec{A}$', Â', $\mathcal{P}$', $\vec{\mathcal{B}}$S') of $\Phi(\vec{A}, \hat{A}, \mathcal{P}, \vec{\mathcal{B}}\text{S}, A_{add})$ is:

- $\vec{A}$':= $\vec{A} \cup [A_{add}]$.
- Â':= (Â − [T]) ∪ *New-Tasks*.
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. adding the new justification specified in the expansion segment to L as the justification of the lowest level of abstraction, and
  2. adding the proof lines *New-Lines*.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

PPLANNER and CPLANNER Actions

A **PPLANNER** or **CPLANNER** action A_S is applicable, if all actions $[A_1,\ldots,A_n]$ in its **action-sequence** slot are applicable when introduced successively. When applied to A_S, Φ_{MULTI} stepwise introduces the actions from the sequence $[A_1,\ldots,A_n]$ using the function $\vec{\Phi}_{\text{MULTI}}$. Afterwards, it replaces $[A_1,\ldots A_n]$ in the constructed action sequence by A_S. That is, the actions $A_1,\ldots,A_n$ are not explicitly mentioned in the constructed action sequence but only implicitly as part of the action of **PPLANNER** or **CPLANNER**. This guarantees that Φ_{MULTI} and $\vec{\Phi}_{\text{MULTI}}$ create a sequence of strategic actions.

Definition 7.13 (Applicable CPLANNER and PPLANNER Actions): Let $\mathcal{P}$ be a $\mathcal{PDS}$, $\mathcal{B}$S a binding store, and A_{add} a **PPLANNER** or **CPLANNER** with the action sequence $[A_1,\ldots,A_n]$. Moreover, let $T_{A_{add}}$ be the task of A_{add} and $\mathcal{B}\text{S}_{A_{add}}$ its binding store. A_{add} is *applicable* with respect to $\mathcal{P}$ and $\mathcal{B}$S, if for each $A_i, i = 1\ldots n$ in $[A_1,\ldots,A_n]$ holds:

- Let $(\vec{A}_i, \hat{\text{A}}_i, \mathcal{P}_i, \vec{\mathcal{B}\text{S}}_i) := \vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}\text{S}}, [A_1,\ldots,A_{i-1}])$ for an arbitrary sequence of actions $\vec{A}$ and an agenda $\hat{\text{A}}$ that contains the task $T_{A_{add}}$. Then, A_i is applicable with respect to $\mathcal{P}_i$, and $\vec{\mathcal{B}\text{S}}_i$ and $\hat{\text{A}}_i$ contains the task of A_i.

□

Definition 7.14 (Φ_{MULTI} on PPLANNER or CPLANNER Actions): Let $\vec{\mathcal{B}\text{S}}$ be a sequence of bindings stores and let $\mathcal{B}$S be the last binding store of $\vec{\mathcal{B}\text{S}}$. Let $\vec{A}$ be a sequence of actions and let A_{add} be a **PPLANNER** or **CPLANNER** action, which is applicable with respect to a $\mathcal{PDS}$ $\mathcal{P}$ and $\mathcal{B}$S.

Moreover, let $[A_1,\ldots,A_n]$ be the action-sequence of A_{add}.
$(\vec{A}_{rec}, \hat{\text{A}}_{rec}, \mathcal{P}_{rec}, \vec{\mathcal{B}\text{S}}_{rec}) := \vec{\Phi}_{\text{MULTI}}(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}\text{S}}, [A_1,\ldots,A_n])$.

If $\hat{\text{A}}$ is an agenda that contains the task of A_{add}, then the result $(\vec{A}', \hat{\text{A}}', \mathcal{P}', \vec{\mathcal{B}\text{S}}')$ of $\Phi(\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}\text{S}}, A_{add})$ is defined by:

- $\vec{A}' := (\vec{A}_{rec} - [A_1,\ldots,A_n]) \cup [A_{add}]$.
- $\hat{\text{A}}' := \hat{\text{A}}_{rec}$.
- $\mathcal{P}' := \mathcal{P}_{rec}$.
- $\vec{\mathcal{B}\text{S}}' := \vec{\mathcal{B}\text{S}}_{rec}$.

□

With the function $\vec{\Phi}_{\text{MULTI}}$ we can define strategic proof plans and strategic solution proof plans. Actually, we shall give three different notions of solution proof plans, which specify more and more strict conditions for strategic proof plans.

Definition 7.15 (Strategic Proof Plans, Strategic Solution Proof Plans): Let $(Thm, \{Ass_1,\ldots,Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$ be a strategic proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and $\hat{\text{A}}_{init}$ its initial agenda.

A *strategic proof plan* to the strategic proof planning problem is a quadruple $\mathsf{SPP} = (\vec{A}, \hat{\text{A}}, \mathcal{P}, \vec{\mathcal{B}\text{S}})$ with a sequence of strategic actions $\vec{A}$, an agenda $\hat{\text{A}}$, a $\mathcal{PDS}$ $\mathcal{P}$, and a sequence of binding stores $\vec{\mathcal{B}\text{S}}$ such that:

1. each strategy of an action of $\vec{A}$ is in $\mathcal{S}$,

2. $(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}\mathrm{S}}) = \vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$,

□

With respect to this definition of a strategic proof plan we can also say that $\Phi_{\textsc{Multi}}$ maps a strategic proof plan and an action into a strategic proof plan and that $\vec{\Phi}_{\textsc{Multi}}$ maps a strategic proof plan and a sequence of strategic actions into a strategic proof plan.

Definition 7.16 (Strategic Solution Proof Plans):
Let $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_{\mathcal{S}})$ be a strategic proof planning problem, $\mathcal{P}_{init}$ the initial $\mathcal{PDS}$ of this problem, and $\hat{\mathrm{A}}_{init}$ its initial agenda.

We distinguish the following three notions of a *strategic solution proof plan*:

- A *method-level solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an agenda without line-tasks and a closed $\mathcal{PDS}$.
- An *instantiated method-level solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an agenda without line-tasks and instantiation-tasks, a closed $\mathcal{PDS}$, and a binding store sequence such that the last binding store contains bindings for all meta-variables occurring in proof lines of the final $\mathcal{PDS}$.
- A *full solution proof plan* for the problem is a sequence of strategic actions $\vec{A}$ such that $\vec{\Phi}_{\textsc{Multi}}([], \hat{\mathrm{A}}_{init}, \mathcal{P}_{init}, [], \vec{A})$ results in an empty agenda, a closed $\mathcal{PDS}$ in which all nodes are justified by ND-rules, and a binding store sequence such that the last binding store contains bindings for all meta-variables occurring in proof lines of the final $\mathcal{PDS}$.

□

The first notion of solution proof plan is called *method-level solution proof plan* since a strategic proof plan satisfying these conditions is reached by computing method actions whose introduction satisfies all line-tasks and creates a closed $\mathcal{PDS}$. Instantiation-tasks and expansion-tasks can be ignored. The second notion of solution proof plan, *instantiated method-level solution proof plan*, demands to tackle also instantiation-tasks. However, expansion-tasks can still be ignored. Finally, in order to obtain a *full solution proof plan* the expansion-tasks have to be solved. We shall describe in section 7.6.2 how a user can make MULTI search for a particular kind of solution proof plan.

7.4 Strategic Manipulation Records

Similar to PLAN, MULTI constructs a history consisting of *manipulation records*. These manipulation records contain information, which can be used by the control rules in order to perform meta-reasoning.

A strategy execution of the algorithms **EXP**, **ATP**, and **INSTMETA** creates one so-called *strategy-application record* (see Figure 7.5). The slots **agenda** and **alternative-job-offers** capture the context in which the manipulation was done whereas the the slots **introduced-action**, **new-tasks**, and **execution-message** store the result of the manipulation. The slot **agenda** captures the agenda before the strategy is applied. The slot **alternative-job-offers** contains the list of alternative job offers, when the strategy

Strategy-Application:	
agenda	
alternative-job-offers	
introduced-action	
new-tasks	
execution-message	

Figure 7.5: A strategy-application record.

was applied. The first job offer in this list is the applied strategy and the task to which the strategy was applied. The performed manipulation, namely the action introduced by the execution of the strategy, is stored in the **introduced-action** slot. This slot is empty, if the execution of a strategy failed. The new tasks created by the introduction of the action are stored in the slot **new-tasks**. The slot **execution-message** contains the execution-message returned by the strategy execution.

Strategy executions of the algorithms **PPlanner** and **CPlanner** create two manipulation records. When they are invoked or re-invoked, they create a *strategy-start record*; when they terminate or are interrupted, then they create a *strategy-stop record*. Figure 7.6 shows the skeletons of these two manipulation records.

Strategy-Start:	
agenda	
alternative-job-offers	
task-tag	

Strategy-Stop:	
task-tag	
introduced-action	
new-tasks	
execution-message	

Figure 7.6: Manipulation records created by **PPlanner** and **CPlanner**.

The strategy-start and strategy-stop records divide the information of a strategy-application record into two parts: the information available when the strategy is invoked or re-invoked, which is stored in a strategy-start record, and the information available when the strategy stops, which is stored in a strategy-stop record. Hence, a strategy-start record has the slots **agenda** and **alternative-job-offers** whereas a strategy-stop record has the slots **introduced-action**, **new-tasks**, and **execution-message**. Additionally, both records have the slot **task-tag**, which contains the task-tag that uniquely identifies the strategy execution.

Note that the manipulation records of the steps performed within a strategy execution of **PPlanner** or **CPlanner** are themselves part of the history. They are not stored in a **PPlanner** or **CPlanner** history element but only delimited by the strategy-start and strategy-stop records of the strategy execution. This approach makes information available as early as possible. In particular, the information on the situation when the strategy was invoked or re-invoked and the information on all steps performed by a strategy execution so far are available for the control rules evaluated within the strategy execution.

Strategies of the **BackTrack** algorithm create two manipulation records whose skeletons are given in Figure 7.7. The *backtrack-start record* contains the information available when the backtracking is started (stored in the **agenda** and **alternative-job-offers** slots) as well as the information which actions the strategy decided to delete. The *backtrack-stop record* contains the information available when the **BackTrack** strategy stops. Since strategies of **BackTrack** do not create actions, this record contains only a slot for the execution message.

BackTrack-Start:	
agenda	
alternative-job-offers	
actions-to-delete	

BackTrack-Stop:	
execution-message	

Figure 7.7: Manipulation records created by **BackTrack**.

Similar to **CPlanner** and **PPlanner**, strategy executions of **BackTrack** successively perform also a set of individual steps. When executed, a strategy of **BackTrack** computes first which actions it has to delete. These actions are stored in the start record. However, in order to delete these actions maybe other actions have to be deleted as well (see section 7.5.7 for details). All single deletion steps are stored in action-deletion records as in PLAN (see section 4.2). Hence, a start and stop record pair of a **BackTrack** strategy execution delimits the manipulation records of all single deletion steps performed within this strategy execution.

7.5 The Algorithms

In this section, we shall describe the algorithms used in Multi. First, we explain Multi's top-level algorithm. Then, we describe the refinement and modification algorithms integrated so far, namely **PPlanner**, **CPlanner**, **Exp**, **ATP**, **InstMeta**, and **BackTrack**.

In the remainder of this section we assume that each function and algorithm used in Multi has access to the blackboards and the entries on them. Hence, when an algorithm or a function accesses information from a blackboard we shall not mention the respective blackboard explicitly as an argument of the function. The only exceptions are the functions *write-onto-blackboard*, which sets the value of an entry on a blackboard, and *take-from-blackboard*, which returns the value of an entry on a blackboard. Both functions obtain the blackboard on which they should work as argument. In the following descriptions of the algorithms we use PB and CB as abbreviations for the proof blackboard and the control-blackboard, respectively.

7.5.1 The Multi Algorithm

Figure 7.8 gives a pseudo-code description of the Multi algorithm. Multi is applied to a strategic proof planning problem with a theorem Thm, a set of assumptions $Ass_1,\ldots,Ass_n$, a set of strategies $\mathcal{S}$, and a set of strategic control rules $\mathcal{C}_\mathcal{S}$. Its output is a strategic proof plan for the given problem $(Thm,\{Ass_1,\ldots,Ass_n\},\mathcal{S},\mathcal{C}_\mathcal{S})$. Multi's first step is to initialize the proof and the control blackboard. It writes onto the proof blackboard an empty sequence of actions, the initial agenda and the initial $\mathcal{PDS}$ of the given problem, and a sequence of binding stores whose only entry consists of an empty binding store. Moreover, it writes onto the control blackboard an empty set of memory entries, an empty set of demands, and an empty sequence of job offers.

The next four steps, steps 2—5 in Figure 7.8, of Multi perform the strategy selection and invokation cycle that is sketched in Figure 6.2 in the previous chapter. Step 2 employs the functions *trigger-jobs-from-strategies* and *trigger-jobs-from-memory*. *trigger-jobs-from-strategies* checks whether the condition of an element of S is satisfied by some tasks of the current agenda on the proof blackboard. A strategy $S \in \mathcal{S}$ places a job offer onto the control blackboard for each task T for which its condition

Input: A strategic proof planning problem $(Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_S)$ with a theorem formula Thm, a set of assumption formulas $Ass_1, \ldots, Ass_n$, a list of strategies $\mathcal{S}$, and a list of strategic control rules $\mathcal{C}_S$.

Output: A strategic proof plan SPP $= (\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{BS})$ with a sequence of strategic actions $\vec{A}$, an agenda $\hat{\mathrm{A}}$, a $\mathcal{PDS}$ $\mathcal{P}$, and a sequence of binding stores $\vec{BS}$.

```
Algorithm: MULTI($Thm, \{Ass_1, \ldots, Ass_n\}, \mathcal{S}, \mathcal{C}_S$)

  1. Initialization
     Let $\hat{\mathrm{A}}$:=initial-agenda($Thm, \{Ass_1, \ldots, Ass_n\}$).
     Let $\mathcal{P}$:=initial-$\mathcal{PDS}$($Thm, \{Ass_1, \ldots, Ass_n\}$).
     write-onto-blackboard([], sequence-of-actions, $PB$).
     write-onto-blackboard($\hat{\mathrm{A}}$, agenda, $PB$).
     write-onto-blackboard($\mathcal{P}$, pds, $PB$).
     write-onto-blackboard([{}], sequence-of-binding-stores, $PB$).
     write-onto-blackboard([], history, $PB$).
     write-onto-blackboard($\emptyset$, memory, $CB$).
     write-onto-blackboard($\emptyset$, demands, $CB$).
     write-onto-blackboard([], job-offers, $CB$).
  2. Job Offers
     trigger-jobs-from-strategies($\mathcal{S}$).
     trigger-jobs-from-memory().
  3. Guidance
     invoke(MetaReasoner, $\mathcal{C}_S$).
  4. Invocation
     Let $\mathcal{J}$:=remove-free-jobs(take-from-blackboard(job-offers, $CB$)).
     If $\mathcal{J} = \emptyset$
       then
         terminate and return
         (take-from-blackboard(sequence-of-actions, $PB$),
          take-from-blackboard(agenda, $PB$),
          take-from-blackboard(pds, $PB$),
          take-from-blackboard(sequence-of-binding-stores, $PB$)).
       else
         Let $J$:=first($\mathcal{J}$).
         If job-offer-from-strategy($J$)
           then (i.e., $J = (S,T)$)
             invoke(algorithm-of-strategy($S$), $(S,T)$, $\mathcal{J}$).
           else (i.e., $J = (@_T, Demands)$)
             invoke(algorithm-of-task-tag($@_T$), $@_T$, $\mathcal{J}$).
  5. Execution
     Wait until strategy-ks-terminated().
  6. Administration
     If strategy-ks-terminated-successful(), then delete-satisfied-demands().
     Goto step 2.
```

Figure 7.8: The MULTI algorithm.

is true. The function *trigger-jobs-from-memory* writes for each memory entry a job offer onto the control blackboard. Afterwards, step 3 invokes the MetaReasoner, which evaluates the strategic control rules $\mathcal{C}_S$ on the job offers.

In step 4, MULTI first reads the resulting list of job offers and deletes the job offers whose strategies have still uninstantiated free parameters. If the resulting list is empty, then MULTI terminates and returns the strategic proof plan (i.e., the sequence of actions, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores) on the proof blackboard. Otherwise MULTI picks the first job offer and invokes the corresponding strategy. If the job offer was placed by a strategy S with respect to a task T, which satisfies the condition of S, then MULTI invokes the algorithm of S with the pair (S,T) as argument. If the job offer was placed from a memory entry with task tag $@_T$, then *algorithm-of-task-tag* computes the algorithm that created the tag $@_T$ using information stored in the history and invokes this algorithm with $@_T$ as argument. In both cases the invoked algorithm obtains the list of all job offers on the control blackboard as second argument.

The invoked algorithm refines or modifies the proof blackboard objects and maybe places demands and a memory entry onto the control blackboard. MULTI waits until the execution of the strategy terminates (see step 5). Then, step 6 checks whether the strategy terminated successfully. This check is performed by the function *strategy-ks-terminated-successful*, which looks up the execution message of the last history on the proof blackboard. If this execution message is a success message, then MULTI employs the function *delete-satisfied-demands* to delete all demands on the control blackboard that are satisfied by the terminated strategy execution as well as all pointers in memory entries to those demands. Afterwards, MULTI restarts its cycle by proceeding with step 2.

We conclude this section with two remarks on the described algorithm:

1. When employing the functions *trigger-jobs-from-memory* (in step 2) and *delete-satisfied-demands* (in step 6) MULTI changes the content of the control blackboard. This is a violation of the blackboard principle, which states that the content of the blackboards should only be changed by respective knowledge sources. For the sake of simplicity of MULTI's blackboard approach we implemented these minor blackboard changes as direct functionalities of the MULTI algorithm. However, in order to avoid a violation of the blackboard principle, we could understand these two functions as particular knowledge sources working on the control blackboard, which are scheduled by MULTI in a predefined way.

2. PLAN terminates either with a solution proof plan or, after traversing the search space, with a failure. MULTI terminates as soon as there is no further job offer to invoke (see step 4). However, the lack of job offers states nothing about the status of the strategic proof planning process. When there are no further tasks in the agenda, then there are no further job offers since there is a strategic solution proof plan on the proof blackboard. But it is possible that there are still tasks in the agenda although there are no further job offers. It is possible that there are no strategies to tackle these tasks (i.e., there is no strategy whose condition is satisfied by the task) or strategic control rules can remove all existing job offers. If MULTI terminates and there are still tasks in the agenda, then it is up to the user to analyze the situation. Is the strategic proof plan created so far a sufficient solution proof plan (when the user is interested in a method-level solution proof plan then expansion-tasks and instantiation-tasks can be ignored)? Are further strategies needed that can deal with particular tasks? Are less restrictive strategic control rules needed that do not remove so much job offers?

7.5.2 The **PPLANNER** Algorithm

Strategies of the algorithm **PPLANNER** refine a strategic proof plan by successively adding method actions, which **PPLANNER** abstracts in one strategic action, when it terminates. A strategy of **PPLANNER** specifies four parameters: a procedure for the computation of the next method action to introduce, parameters for the set of usable methods and control rules, and a termination condition. We discussed some strategies of **PPLANNER** already in section 6.2.1. More examples are given in the following chapters, when we describe the case studies.

Figure 7.9 gives a pseudo-code description of the **PPLANNER** algorithm. **PPLANNER** obtains two arguments. When a **PPLANNER** strategy S is intially invoked, then **PPLANNER**'s first input is a pair (S, T) consisting of the strategy S and a line-task T. When a strategy execution is re-invoked, then the first argument is the task tag of the strategy execution. The second argument for **PPLANNER** is the list of all alternative job offers on the control blackboard, when **PPLANNER** is invoked. **PPLANNER** returns no specific output but updates the content of the proof blackboard by introducing successively method actions. Essentially, **PPLANNER** performs a cycle of task selection, action selection, and action introduction, which is similar to the cycle of PLAN. This core cycle is completed by an initialization step and different events that stop the **PPLANNER** algorithm, namely successful termination, interruption, and failure.

In the initialization step (step 1 in Figure 7.9) **PPLANNER** extracts the information of the strategy and the initial task with respect to which it runs. First, it employs the function *extract-from-input*, which computes the current task tag $@_T$, the current strategy S, and the initial task T. If the first input of **PPLANNER** is a pair (S, T) (i.e., initial call of S on T), then the information on S and T is directlty accessible and *extract-from-input* creates a new task tag $@_T$, which it attaches to T. If the first input of **PPLANNER** is a task tag $@_T$ (i.e., re-invokation of interrupted strategy execution), then *extract-from-input* employs information from the history to compute the strategy S and the initial task T that correspond to the given task tag. Next, **PPLANNER** uses the function *parameters-of-strategy* to obtain the parameters of the strategy S, which are a list of methods $\mathcal{M}$, a list of control rules $\mathcal{C}$, the termination condition, and the action computation and selection procedure. So far, we have implemented two action computation and selection procedures, namely **CHOOSEACTION** (see section 4.2.4) and **CHOOSEACTIONALL** (see appendix A).[5] Afterwards, **PPLANNER** adds a strategy-start record to the history and sets the algorithm variable $\vec{A}_{add}$ to the empty list. In this variable **PPLANNER** stores the method actions, which it introduces successively.

Step 2 and step 3 in Figure 7.9 check whether **PPLANNER** terminates successfully or interrupts. We postpone the detailed discussion of these two steps until the discussion of step 7 in order to discuss together all three steps that stop **PPLANNER** and the differences among them. The next three steps — step 4, step 5, and step 6 — are the core cycle of selecting the next task, computing and selecting the next method action, and introducing the selected action. Essentially, these steps correspond to step 2, step 3, and step 4 of PLAN in Figure 4.9 in section 4.2.2, they are only sligthly adapted to MULTI. When **PPLANNER** selects the next task to tackle in step 4, then it evaluates the control rules of kind 'Task' not on the whole agenda of the

[5] Note that parts of these algorithms work slightly differently when used in MULTI as opposed to the functionality described in section 4.2.4 and appendix A. All functions used within these algorithms that match proof lines of a method with proof lines of a task (e.g., *match-task-line*, *match-s+p* see section 4.2.4) apply first the bindings of the current binding store to the proof lines of the task. Then, they perform the respective matchings with respect to this "up-to-date" proof lines instead of the original ones.

Input: (1) either a pair (S, T) where S is a PPLANNER strategy and T is a line-task or a task tag $@_T$, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: PPLANNER($arg_1$, $\mathcal{J}_{rest}$))

 1. Initialization
    Let $(@_T, S, T)$:=extract-from-input($arg_1$).
    Let $(\mathcal{M}, \mathcal{C}, term\text{-}cond, action\text{-}proc)$:=parameters-of-strategy($S$).
    add-strategy-start-record-to-history($\mathcal{J}_{rest}$, $@_T$).
    Let $\vec{A}_{add}$:=[].

 2. Successful Termination Check
    (see Figure 7.10)

 3. Interruption Check
    (see Figure 7.10)

 4. Task Selection:
    Let current task $T_{curr}$:= first(evalcrules-tasks(tasks-with-tag($@_T$), $\mathcal{C}$)).

 5. Action Selection
    Let ($A_{add}$,$\mathcal{A}$):=apply($action\text{-}proc, T_{curr}, \mathcal{M}, \mathcal{C}$) where $A_{add}$ is an action
    and $\mathcal{A}$ is the set of computed alternative actions.

 6. Action Introduction
    If $A_{add}$ is given
       then
          PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.
          add-action-intro-record($A_{add}$,$\mathcal{A}$).
          $\vec{A}_{add}$:=$\vec{A}_{add} \cup [A_{add}]$.
          If extract-constraints($A_{add}$) $\neq \emptyset$
             then
                pass-constraints(extract-constraints($A_{add}$)).
          Goto step 2.

 7. Failure
    (see Figure 7.10)
```

Figure 7.9: The PPLANNER algorithm.

proof blackboard, but only on the tasks that carry the current task tag $@_T$ (the restricted initial alternative list is computed by the function *tasks-with-tag*). Whereas in PLAN the application of the algorithm CHOOSEACTION is fix, PPLANNER applies the action computation procedure specified as parameter of the current strategy in step 5. When an action is found, then PPLANNER applies this action in step 6 with the function Φ_{MULTI} to the action sequence, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores on the proof blackboard. We write this as "PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$" and do not refer to the changed elements of the proof blackboard explicitly. Similar to PLAN, PPLANNER adds a history entry for the introduced action and passes new constraints to external constraint solvers. Additionally, the introduced action is added to $\vec{A}_{add}$. Afterwards, PPLANNER continues with step 2.

PPLANNER can stop at three different places, namely step 2, step 3 and step 7,

```
2. Termination Check
   If no-tasks-with-tag($@_T$) or apply($term\text{-}cond$) = $true$
     then
       Let $message$:=create-success-message($S,T$).
       Let $A^S_{add}$:=create-strategic-action($\vec{A}_{add}$).
       replace-actions($\vec{A}_{add}, A^S_{add}$).
       remove-tag($@_T$).
       add-strategy-stop-record-to-history($@_T, A^S_{add}, message$).
       Terminate.

3. Interruption Check
   Let $I$:=first(evalcrules-interrupt([Nil, True], $\mathcal{C}$)).
   If $I$ = True
     then
       Let $message$:=create-interrupt-message($S,T$).
       Let $A^S_{add}$:=create-strategic-action($\vec{A}_{add}$).
       replace-actions($\vec{A}_{add}, A^S_{add}$).
       write-to-demands(demands($I$)).
       write-to-memory($@_T$, demands($I$)).
       add-strategy-stop-record-to-history($@_T, A^S_{add}, message$).
       Terminate.

7. Failure
   IF $A_{add}$ is not given
     then
       Let $message$:=create-failure-message($S,T$).
       Let $A^S_{add}$:=create-strategic-action($\vec{A}_{add}$).
       replace-actions($\vec{A}_{add}, A^S_{add}$).
       write-to-demands($\{? - ON - T\}$).
       write-to-memory($@_T, \{? - ON - T\}$).
       add-strategy-stop-record-to-history($@_T, A^S_{add}, message$).
       Terminate.
```

Figure 7.10: Leaving the **PPLANNER** algorithm.

which are given in detail in Figure 7.10. Step 2 checks whether the application of the strategy of **PPLANNER** was successful such that **PPLANNER** should stop. This is the case either when the termination condition of the strategy is satisfied or when there are no further tasks which carry the task tag of the strategy execution. Step 3 employs the function *evalcrules-interrupt* to evaluate the control rules of kind ‘Interrupt’ on the alternative list [**False**,**True**], where **False** causes no interrupt whereas **True** causes an interrupt. The control rules of kind ‘Interrupt’ can also compute demands and attach the demands to the **True** element of the alternative list. Finally, step 7 is performed, when step 5 does not provide a method action to introduce, that is, step 7 deals with a failure situation in **PPLANNER**.

Some computations are the same in all three steps. They all compute an execution message $message$ and employ the function *create-strategic-action* to compute a strategic action A^S_{add} from the collected sequence of method actions $\vec{A}_{add}$. Moreover, they all replace the sequence of method actions by a new strategic action in the action sequence on the proof blackboard (this is done by the function *replace-actions*).

Finally, they all add a strategy-stop entry to the history before they terminate. The three steps differ in the created execution message and in whether and which memory entries and demands they create. When the strategy knowledge source terminates successfully, then **PPlanner** creates a success message and does not write memory entries or demands onto the control blackboard. Rather, it applies the function *remove-tag*, which removes its task tag from all tasks in the agenda on the proof blackboard. If the execution of the strategy interrupts, then it creates an interruption message and places a memory entry and demands onto the control blackboard. The demands stem from the evaluated control rules of kind 'Interrupt' and the memory entry consists of the task tag and pointers to the added demands. If **PPlanner** has to deal with a failure occuring with respect to the task T_{curr}, then it creates a failure message. Moreover, it writes a task-demand $? - ON - T_{curr}$ and a memory entry consisting of the task tag and a pointer to this task-demand onto the control blackboard. Since a failure creates a memory entry and a demand, we can understand it as a special kind of interrupt — the difference with respect to the origin of the interruption is recorded in the execution messages.

The further interpretation of and reaction to the termination is left to Multi and meta-reasoning at the strategy-level (this holds also for all other refinement and modification algorithms employed by Multi, which can terminate in different ways). If the last strategy execution terminated with a success message, then Multi deletes all demands on the control blackboard that are satisfied by this strategy execution (see previous section). Moreover, strategic control rules can make use of the information contained in the execution messages. For instance, the strategic control rule `prefer-backtrack-if-failure` (see section 6.2.3) analyses the execution messages and prefers to perform some backtracking if the last strategy was a **PPlanner** strategy and terminated with a failure message. This control rule (which can be overwritten by more specific control rules) forces a systematic traversal of the search space given by a **PPlanner** strategy.

7.5.3 The **CPlanner** Algorithm

Strategies of the algorithm **CPlanner** refine a strategic proof plan by successively transfering actions from a source proof plan into the proof plan under construction. A strategy of **CPlanner** specifies three parameters: a list of action transfer procedures, a list of control rules, and a termination condition. We discussed an example strategy of **CPlanner** already in section 6.2.4. More examples are discussed in [210].

Figure 7.11 gives a pseudo-code description of **CPlanner**. **CPlanner** obtains two arguments. When a **CPlanner** strategy S is intially invoked, then **CPlanner**'s first input is a pair (S,T) consisting of the strategy S and a line-task T. When a strategy execution is re-invoked, then the first argument is the task tag of the strategy execution. The second argument for **CPlanner** is the list of all alternative job offers on the control blackboard, when **CPlanner** is invoked. **CPlanner** returns no specific output but updates the content of the proof blackboard by introducing successively method actions.

Several parts of the **CPlanner** algorithm are equal or similar to the **PPlanner** algorithm. As **PPlanner** **CPlanner** starts with the extraction of the strategy information and the initial task in step 1. In particular, step 1 extracts the action transfer procedures $\mathcal{TP}$ and sets the algorithm variable $\vec{A}_{add}$ to the empty list. In this variable **CPlanner** stores the actions, which it introduces successively. Afterwards, step 2 and step 3 check whether **CPlanner** terminates successfully or interrupts. These two steps equal step 2 and step 3 of **PPlanner**, respectively, given in Figure 7.10.

Input: (1) either a pair (S,T) where S is a **CPLANNER** strategy and T is a task or a task tag $@_T$,
(2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: CPLANNER($arg_1, \mathcal{J}_{rest}$))

  1. Initialization
     Let $(@_T, S, T)$:=extract-from-input$(arg_1)$.
     Let $(\mathcal{TP}, \mathcal{C}, term\text{-}cond)$:=parameters-of-strategy$(S)$.
     add-strategy-start-record-to-history$(\mathcal{J}_{rest}, @_T)$.
     Let $\vec{A}_{add}$:=[].

  2. Successful Termination Check
     (see PPLANNER Figure 7.10)

  3. Interruption Check
     (see PPLANNER Figure 7.10)

  4. Select and Evaluate Transfer Procedures
     Let $\mathcal{TP}_{rest}$:=evalcrules-transferprocs$(\mathcal{TP})$.
     Until $(Obj$ is action or demand) or $(\mathcal{TP}_{rest} = [])$
          Let $TP_{curr}$:=first$(\mathcal{TP}_{rest})$.
          Let $Obj$:=evaluate$(TP_{curr})$.
          $\mathcal{TP}_{rest}$:=rest$(\mathcal{TP}_{rest})$.

  5. Action Introduction
     If $Obj$ is action $A_{add}$
       then
          PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.
          add-action-intro-record$(A_{add}, \mathcal{A})$.
          $\vec{A}_{add}$:=$\vec{A}_{add} \cup [A_{add}]$.
          If extract-constraints$(A_{add}) \neq \emptyset$
            then
               pass-constraints(extract-constraints$(A_{add})$).
          Goto step 2.

  6. Demand Interruption
     If $Obj$ is demand $D_{add}$
       then
          Let $message$:=create-interrupt-message$(S,T)$.
          Let $A^S_{add}$:=create-strategic-action$(\vec{A}_{add})$.
          replace-actions$(\vec{A}_{add}, A^S_{add})$.
          write-to-demands$(D_{add})$.
          write-to-memory$(@_T, D_{add})$.
          add-strategy-stop-record-to-history$(@_T, A^S_{add}, message)$.
          Terminate.

  7. Failure
     (see PPLANNER Figure 7.10)
```

Figure 7.11: The **CPLANNER** algorithm.

Step 4 first evaluates the control rules of kind 'TransferProcedure' on the alternative action transfer procedures $\mathcal{TP}$. This results in a changed and re-ordered alternative list $\mathcal{TP}_{rest}$. Then, step 4 evaluates the action transfer procedures in the order of this list until either one procedure provides an action or a demand, which is stored in the algorithm variable Obj, or all procedures have been tried. That is, at the end of step 4 Obj is either bound to an action A_{add} or to a demand D_{add} or it is unbound. These three cases are covered by the following steps, respectively. Step 5 describes the processing of an action A_{add}. In this case, **CPlanner** introduces A_{add} into the proof plan under construction employing the function $\Phi_{\textsc{Multi}}$. Moreover, it adds a history entry for the introduced action and passes new constraints to external constraint solvers. Additionally, the introduced action is added to $\vec{A}_{add}$. Then, **CPlanner** continues with step 2. Step 6 processes a demand D_{add}. It writes the demand onto the control blackboard and terminates then with an interrupt message. If the evaluation of the action transfer procedure provides neither an action nor a demand, then **CPlanner** terminates in step 7 with a failure message. This step equals step 7 of **PPlanner** in Figure 7.10.

7.5.4 The **InstMeta** Algorithm

Strategies of the algorithm **InstMeta** tackle an instantiation-task and compute a binding for the meta-variable of the instantiation-task. With this new binding a new binding store is created, which is added to the sequence of binding stores on the proof blackboard. A strategy of **InstMeta** specifies one parameter, namely a function that determines how the instantiation for a meta-variable is computed. We discussed some strategies of **InstMeta** in section 6.2.1. More examples are given in the following chapters, when we describe the case studies.

Figure 7.12 contains a pseudo-code description of **InstMeta**. **InstMeta** has two arguments. First, a pair (S,T), which consists of an **InstMeta** strategy S and an instantiation-task T. Second, the list of all alternative job offers on the control blackboard, when the **InstMeta** strategy was invoked. **InstMeta** returns no specific output but updates the content of the proof blackboard.

Step 1 in Figure 7.12 applies the instantiation computation function of the strategy S to the task T. This function application can either succeed or fail. If the function application succeeds, then the algorithm variable *inst* is bound to the returned value. Otherwise *inst* stays unbound. Step 2 computes an instantiation action when *inst* is bound and applies this action with $\Phi_{\textsc{Multi}}$ to the strategic proof plan elements on the proof blackboard. Finally, step 3 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether *inst* is bound or not. When *inst* is bound **InstMeta** creates a success message, otherwise **InstMeta** creates a failure message.

Currently, the computation function of an **InstMeta** strategy is provides either one (success) or no (failure) solution. This was sufficient for the case studies conducted so far. When it turns out that a set of alternative instantiations and reasoning on the selection of one alternative is needed, then **InstMeta** can easily be extended to cover this functionality: The variable *inst* has to store a list of alternatives. Moreover, between step 1 and step 2 an additional step is needed, which evaluates control rules on the alternative instantiations and selects one. The control rules would become an additional parameter of **InstMeta**.

Input: (1) a pair (S,T) where S is a **INSTMETA** strategy and T is an instantiation-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: INSTMETA($(S,T),\mathcal{J}$)

  1. Compute Instantiation
     Let $inst$:=apply(compute-inst-function($S$),$T$).

  2. Compute and Apply Action
     If bound($inst$)
        then
           Let $A_{add}$:=new-instmeta-action($S,T,inst$).
           PB:=$\Phi_{\text{MULTI}}(A_{add},PB)$.

  3. Update History
     If bound($inst$)
        than
           Let $message$:=create-success-message($S,T$).
           add-strategy-application-record-to-history($\mathcal{J},A_{add},\emptyset,message$).
        else
           Let $message$:=create-failure-message($S,T$).
           add-strategy-application-record-to-history($\mathcal{J},\emptyset,\emptyset,message$).
     Terminate.
```

Figure 7.12: The **INSTMETA** algorithm.

7.5.5 The ATP Algorithm

Strategies of the algorithm **ATP** refine a strategic proof plan by solving a line-task with an **ATP** action. They apply external automated theorem provers and check whether their output is a proof. A strategy of **ATP** specifies two parameters for these two functionalities, namely an application function and an output check function. We discussed a strategy of **ATP** in section 6.2.4. More examples are given in the following chapters, when we describe the case studies.

Figure 7.13 contains a pseudo-code description of the **ATP** algorithm. **ATP** has two arguments. First, a pair (S,T), which consists of an **ATP** strategy S and an instantiation-task T. Second, the list of all alternative job offers on the control blackboard, when the **ATP** strategy was invoked. **ATP** returns no specific output but updates the content of the proof blackboard.

Step 1 applies the application function of the strategy S to the task T. This function application provides an output, which is stored in the algorithm variable *out*. Step 2 applies the output check function to *out*, which returns either *true* or *nil*. If the result, which is stored in the algorithm variable *check*, is *true*, then *out* is accepted as proof. In this case, **ATP** computes an action and applies this action with Φ_{MULTI} to the strategic proof plan elements on the proof blackboard (see step 3 in Figure 7.13). Finally, step 4 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether *check* is *true*. If *check* is *true*, then **ATP** creates a success message, otherwise it creates a failure message.

Input: (1) a pair (S, T) where S is an **ATP** and T is a line-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: ATP((S,T), 𝒥)

  1. Apply Provers
     Let $out$:=apply(atp-apply-function(S), T).

  2. Check Output
     Let $check$:=apply(atp-output-check-function(S), out, T).

  3. Compute and Apply Action
     If $check = true$
       then
         Let $A_{add}$:=new-atp-action(S, T, out).
         PB:=$\Phi_{\textsc{Multi}}(A_{add}, PB)$.

  4. Update History
     If $check = true$
       then
         Let $message$:=create-success-message(S, T).
         add-strategy-application-record-to-history($\mathcal{J}, A_{add}, \emptyset, message$).
       else
         Let $message$:=create-failure-message(S, T).
         add-strategy-application-record-to-history($\mathcal{J}, \emptyset, \emptyset, message$).
     Terminate.
```

Figure 7.13: The **ATP** algorithm.

7.5.6 The **Exp** Algorithm

The algorithm **Exp** refines a strategic proof plan by expanding complex steps. When applied to a closed proof line L whose justification is $(J\ P_1\ \ldots\ Pn)$, then **Exp** computes a proof segment that derives L from $P_1, \ldots, P_n$ at a lower level of abstraction. **Exp** has no parameters. The only strategy of **Exp** is ExpS.

Figure 7.14 contains a pseudo-code description of the **Exp** algorithm. **Exp** obtains two arguments. First, a pair (S, T), which consists of a **Exp** strategy S (i.e., ExpS) and an expansion-task T. Second, the list of all alternative job offers on the control blackboard, when the **Exp** strategy was invoked. **Exp** returns no specific output but updates the content of the proof blackboard.

Step 1 tests whether the justification **Exp** should expand is a tactic application or a method application. Depending on what kind of step it finds **Exp** employs either the function *expand-tactic* or the function *expand-method* to compute the expansion proof segment. *expand-tactic* evaluates the expansion procedure of the found tactic whereas *expand-method* instantiates the proof schema of the found method. When these function applications succeed, then the algorithm variable *exp-segment* is bound to the computed proof segment. Otherwise *exp-segment* stays unbound. When *exp-segment* is bound, Step 2 creates an expansion action and applies the action with $\Phi_{\textsc{Multi}}$ to the elements of the strategic proof plan on the proof blackboard. Afterwards, step 3 adds a new strategy-application record to the history on the proof blackboard. The execution message of this record entry depends on whether *exp-segment* is bound or not. When *exp-segment* is bound

Input: (1) a pair (S, T) where S is an **EXP** strategy and $T = L|^{Exp}$ is an expansion-task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: EXP((S,T),J)

1. Compute Expansion-Segment
   Let $(J\ P_1\ \ldots\ P_n)$ be the justification of $L$.
   If is-tactic($J$)
      then
         Let $exp\text{-}segment$:=expand-tactic($L$).
      else
         Let $exp\text{-}segment$:=expand-method($L$).

2. Compute and Apply Action
   If bound($exp\text{-}segment$)
      then
         Let $A_{add}$:=new-expansion-action($S, T, exp\text{-}segment$).
         PB:=$\Phi_{\text{MULTI}}(A_{add}, PB)$.

3. Update History
   If bound$exp\text{-}segment$
      then
         Let $message$:=create-success-message($S, T$).
         add-strategy-application-record-to-history($\mathcal{J}, A_{add}, \emptyset, message$).
      else
         Let $message$:=create-failure-message($S, T$).
         add-strategy-application-record-to-history($\mathcal{J}, \emptyset, \emptyset, message$).
   Terminate.
```

Figure 7.14: The **EXP** algorithm.

EXP creates a success message, otherwise **EXP** creates a failure message.

7.5.7 The BACKTRACK Algorithm

BACKTRACK is an algorithm that removes the actions introduced by other algorithms of MULTI from a strategic proof plan. **BACKTRACK** adds no own actions but only history entries. When to backtrack and which actions to backtrack is not hard-wired in the MULTI algorithm but is subject of the different strategies of **BACKTRACK** and the guidance by reasoning at the strategy-level. A strategy of **BACKTRACK** specifies a function that selects the set of actions in the current strategic proof plan that should be deleted. When MULTI invokes a **BACKTRACK** strategy, then **BACKTRACK** removes all actions explicitly selected by this function as well as all actions that depend from these actions. Thus, the backtracking in MULTI is dependency-directed in the sense discussed in section 4.2. We described a strategy of **BACKTRACK** in section 6.2.1. More examples are given in the following chapters, when we describe the case studies.

Before we give a pseudo-code description of the **BACKTRACK** algorithm we shall introduce the notion of dependency among actions and when an action is deletable.

Both notions are extensions of the concepts introduced for PLAN in section 4.2.3. When an action is introduced into a strategic proof plan, then it modifies the elements of the strategic proof plan. Other actions introduced later on may depend on these modifications. For instance, when a method action introduces a new proof line, which is used lateron by another action, then the second action is not possible without the first action. In the following definition, we shall define for the different kinds of strategic actions and for method actions which other actions in an action sequence depend on them.

Definition 7.17 (Dependent Actions): Let $\vec{A}$ be a sequence of actions with $\vec{A}$=$[A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n]$. The set of actions in $\vec{A}$, which *depend* on A_i is defined for the different kinds of actions in MULTI as follows.

Method Action: Let A_i be a method action with the $\ominus$ conclusions $\ominus Concs$, the $\oplus$ conclusions $\oplus Concs$, and the $\oplus$ premises $\oplus Prems$. If A_i contains some binding constraints, then $\{A_{i+1}, \ldots, A_n\}$ depend on A_i. Otherwise, $A_j \in \{A_{i+1}, \ldots, A_n\}$ depends on A_i if:

1. A_j is a method action whose sets of conclusions or premises contains a proof line of $\oplus Concs$ or $\oplus Prems$ (which are the new proof lines introduced by A_i),
2. A_j is an **INSTMETA** action, which tackles an instantiation-task whose meta-variable is introduced by A_i,
3. A_j is an **EXP** action, which tackles an expansion-task whose proof line is in $\ominus Concs$ or $\oplus Concs$ (the proof lines closed by A_i),
4. A_j is an **ATP** action, which tackles a line-task that contains either as support or as conclusion a proof line of $\oplus Concs$ or $\oplus Prems$,
5. A_j is a **PPLANNER** or **CPLANNER** action, which contains an action that depends on A_i.

INSTMETA Action: Let A_i be an **INSTMETA** action. Then $\{A_{i+1}, \ldots, A_n\}$ depend on A_i.

ATP Action: Let A_i be an **ATP** action. $A_j \in \{A_{i+1}, \ldots, A_n\}$ depends on A_i if A_j is an **EXP** action, which tackles the expansion-task with the proof line closed by A_i.

EXP Action: Let A_i be an **EXP** action with the set $\mathcal{L}_{new}$ of new proof lines in the proof-segment. Let $T = L|^{Exp}$ be the task of A_i. Then $A_j \in \{A_{i+1}, \ldots, A_n\}$ depends on A_i if

1. A_j is a method action, which contains either as conclusion or as premise a proof line of $\mathcal{L}_{new}$, or which contains L as $\ominus$ conclusion,[6]
2. A_j is an **INSTMETA** action, which tackles an instantiation-task whose meta-variable is introduced by A_i,
3. A_j is an **EXP** action, which tackles an expansion-task whose proof line is in $\mathcal{L}_{new}$,
4. A_j is an **ATP** action, which tackles a line-task that contains a proof line of $\mathcal{L}_{new}$ either as support or as goal, or which tackles a line-task whose goal is L,[6]
5. A_j is a **PPLANNER** or **CPLANNER** action, which contains an actions that depends on A_i.

[6] If A_i opens L again, then L can be closed again later on by another method action.

CPlanner or PPlanner Action: Let A_i be a **CPlanner** or a **PPlanner** action whose sequence of actions is $[A'_1, \dots, A'_m]$. Then $A_j \in \{A_{i+1}, \dots, A_n\}$ depends on A_i if there is an action $A'_k \in [A'_1, \dots, A'_m]$ such that A_j depends on A'_k.

Finally, we have to define which actions of an action sequence depend on an action that is contained within a **CPlanner** or **PPlanner** action:
Let A_i be a **CPlanner** or **PPlanner** action whose action sequence is $[A'_1, \dots, A'_{i-1}, A'_i, A'_{i+1}, \dots, A'_n]$. Then the set of actions that depend on A'_i with respect to $\vec{A}$ is the set of actions that depend on A'_i with respect to the action sequence $[A_1, \dots, A_{i-1}] \cup [A'_1, \dots, A'_{i-1}, A'_i, A'_{i+1}, \dots, A'_n] \cup [A_{i+1}, \dots, A_n]$. □

Note that with this definition all actions succeeding an action that introduces new bindings (i.e., method actions with bindings and **InstMeta** actions) depend on this action. We use now the notion of dependency of actions to define when an action is deletable with respect to an action sequence.

Definition 7.18 (Deletable Actions): Let $\vec{A}$ be a sequence of actions with $\vec{A}$=$[A_1, \dots, A_{i-1}, A_{del}, A_{i+1}, \dots, A_n]$. A_{del} is *deletable* with respect to $\vec{A}$ if the set of actions in $\vec{A}$ that depend on A_{del} is empty. □

Next, we define the functions $\Phi^{-1}_{\textsc{Multi}}$ and $\vec{\Phi}^{-1}_{\textsc{Multi}}$, which delete actions.[7] We give first the general outline of $\Phi^{-1}_{\textsc{Multi}}$ and define the recursive $\vec{\Phi}^{-1}_{\textsc{Multi}}$. Afterwards, we define $\Phi^{-1}_{\textsc{Multi}}$ for the different kinds of actions.

Definition 7.19 (Action Deletion Functions $\Phi^{-1}_{\textsc{Multi}}$ and $\Phi_{\vec{\textsc{Multi}}}^{-1}$): The *action deletion function* $\Phi^{-1}_{\textsc{Multi}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores and an action into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\Phi^{-1}_{\textsc{Multi}} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\mathrm{S} \times A_{del} \mapsto \vec{A}\text{'} \times \hat{\mathrm{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\mathrm{S}\text{'}.$$

The *recursive action deletion function* $\vec{\Phi}^{-1}_{\textsc{Multi}}$ is a partial function that maps a sequence of actions, an agenda, a $\mathcal{PDS}$, a sequence of binding stores, and a sequence of actions into a sequence of actions, an agenda, a $\mathcal{PDS}$, and a sequence of binding stores, i.e.,

$$\vec{\Phi}^{-1}\mathrm{MULTI} : \vec{A} \times \hat{\mathrm{A}} \times \mathcal{P} \times \vec{\mathcal{B}}\mathrm{S} \times \vec{A}_{del} \mapsto \vec{A}\text{'} \times \hat{\mathrm{A}}\text{'} \times \mathcal{P}\text{'} \times \vec{\mathcal{B}}\mathrm{S}\text{'}.$$

$\vec{\Phi}^{-1}_{\textsc{Multi}}$ is recursively defined as follows.
Let $\vec{A}$ be a sequence of actions, $\hat{\mathrm{A}}$ an agenda, $\mathcal{P}$ a $\mathcal{PDS}$, $\vec{\mathcal{B}}\mathrm{S}$ a sequence of binding stores, and $\vec{A}_{del}$ a sequence of actions.

1. If $\vec{A}_{del}$ is empty, then
$\vec{\Phi}^{-1}_{\textsc{Multi}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, \vec{A}_{del}) := (\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S})$.

2. Otherwise let A_{del} := *first*$(\vec{A}_{del})$ and $\vec{A}\text{'}_{del}$:= *rest*$(\vec{A}_{del})$. If A_{del} is in $\vec{A}$ or part of a **CPlanner** or **PPlanner** action in $\vec{A}$ and A_{del} is deletable with respect to $\vec{A}$, then
$\vec{\Phi}^{-1}_{\textsc{Multi}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, \vec{A}_{del}) := \vec{\Phi}^{-1}_{\textsc{Multi}}(\Phi^{-1}_{\textsc{Multi}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, A_{del}), \vec{A}\text{'}_{del})$.

□

[7] Since action deletion is conceptually the inverse operation of action introduction we call these functions $\Phi^{-1}_{\textsc{Multi}}$ and $\vec{\Phi}^{-1}_{\textsc{Multi}}$ although technically they are not the inverse functions of $\Phi_{\textsc{Multi}}$ and $\vec{\Phi}_{\textsc{Multi}}$.

In the single definitions of the function $\Phi^{-1}_{\textsc{Multi}}$ for the different kinds of actions we describe the modifications of the sequence of actions, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores caused by the deletion of a respective action. Although the notion of deletability of an action is only defined with respect to a sequence of actions, we assume that the agenda, the $\mathcal{PDS}$, and the sequence of binding stores are not arbitrary, but created by this sequence of actions (in particular, by the action that should be deleted).

We start with the definition of $\Phi^{-1}_{\textsc{Multi}}$ for method actions. Since in Multi the action sequences consist only of strategic actions, a method action can occur only within a **PPlanner** or **CPlanner** action. Hence, the following definition describes the deletion of a method action within a **PPlanner** or **CPlanner** action.

Definition 7.20 ($\Phi^{-1}_{\textsc{Multi}}$ on Method Actions): Let $\vec{A}$ be a sequence of actions and let A_{del} be a method action, which is in an **PPlanner** or **CPlanner** action $A_{planner}$ in $\vec{A}$, i.e., $\vec{A}$=$[A_1,\ldots,A_{i-1},A_{planner},A_{i+1},\ldots,A_n]$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and $\hat{\mathrm{A}}$ an agenda. Moreover, let $\oplus$*Concs* be the $\oplus$ conclusions, $\ominus$*Concs* the $\ominus$ conclusions, $\oplus$*Prems* the $\oplus$ premises, $\ominus$*Prems* the $\ominus$ premises, and *BPrems* the blank premises of A_{del}. Let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del} and let σ be the binding constraints of A_{del}.
Prems:=$\oplus$*Prems* $\cup$ $\ominus$*Prems* $\cup$ *BPrems*,
Concs:=$\oplus$*Concs* $\cup$ $\ominus$*Concs*
Lines-To-Remove:=$\oplus$*Concs* $\cup$ $\oplus$*Prems*
Old-Line-Tasks:=$[L' \blacktriangleleft \mathcal{SUPPS}_{L'} \mid L' \in \oplus Prems]$.
Old-Inst-Tasks:=$[mv|^{Inst} \mid mv \in$ *New-Lines and nowhere else in* $\mathcal{P}]$.
Old-Exp-Tasks:=$[C|^{Exp} \mid C$ *in Concs*$]$.
Tasks-To-Remove:= *Old-Line-Tasks* $\cup$ *Old-Inst-Tasks* $\cup$ *Old-Exp-Tasks.*
New-Inst-Tasks:=$[mv|^{Inst} \mid mv$ *bound in*$\sigma]$.
New-Tasks:=$[T]$ $\cup$ *New-Inst-Tasks.*

If A_{del} is deletable with respect to $\vec{A}$ and if $\hat{\mathrm{A}}$, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result ($\vec{A}$',$\hat{\mathrm{A}}$',$\mathcal{P}$',$\vec{\mathcal{B}}$S') of $\Phi^{-1}_{\textsc{Multi}}(\vec{A},\hat{\mathrm{A}},\mathcal{P},\vec{\mathcal{B}}\mathrm{S},A_{del})$ is defined by:

- $\vec{A}$':= $[A_1,\ldots,A_{i-1},A'_{planner},A_{i+1},\ldots,A_n]$
  where $A'_{planner}$ results from $A_{planner}$ by removing A_{del} from the sequence of actions of $A_{planner}$.
- $\hat{\mathrm{A}}$':= *New-Tasks* $\cup$ ($\hat{\mathrm{A}}$ $-$ *Tasks-To-Remove*).
- $\mathcal{P}$' results from $\mathcal{P}$ by
  1. removing the lines *Lines-To-Remove* and
  2. justifying the proof lines $\ominus$*Concs* with *Open*, respectively.
- If σ is empty, then $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S, otherwise $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S $-$ *last*($\vec{\mathcal{B}}$S).[8]

□

Definition 7.21 ($\Phi^{-1}_{\textsc{Multi}}$ on InstMeta Actions): Let $\vec{A}$ be a sequence of actions and let A_{del} be an **InstMeta** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and $\hat{\mathrm{A}}$ an agenda.

[8] If σ is not empty, then the last binding store in $\vec{\mathcal{B}}$S has to be the binding store resulting from the introduction of A_{del} since otherwise A_{del} would not be deletable. Thus, when A_{del} is deleted, then the last binding store has to be removed.

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result ($\vec{A}$',Â',$\mathcal{P}$',$\vec{\mathcal{B}}$S') of $\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{A}, \mathcal{P}, \vec{\mathcal{B}}S, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A}$ − A_{del}.
- Â':=Â ∪ $[T]$ where T is the task of A_{del}.
- $\mathcal{P}$':=$\mathcal{P}$.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S − *last*($\vec{\mathcal{B}}$S).

□

Definition 7.22 (Φ^{-1}_{MULTI} on ATP Actions): Let $\vec{A}$ be a sequence of actions and let A_{del} be an **ATP** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Let $T = L \blacktriangleleft \mathcal{SUPPS}_L$ be the task of A_{del}.

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result ($\vec{A}$',Â',$\mathcal{P}$',$\vec{\mathcal{B}}$S') of $\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{A}, \mathcal{P}, \vec{\mathcal{B}}S, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A}$ − A_{del}.
- Â':=(Â ∪ $[T]$) − $L|^{Exp}$.
- $\mathcal{P}$' results from $\mathcal{P}$ by opening the line L.
- $\vec{\mathcal{B}}$S':=$\vec{\mathcal{B}}$S.

□

Definition 7.23 (Φ^{-1}_{MULTI} on EXP Actions): Let $\vec{A}$ be a sequence of actions and let A_{del} be an **EXP** action in $\vec{A}$. Let $\vec{\mathcal{B}}$S be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and Â an agenda. Moreover, let $T = L|^{Exp}$ be the task of A_{del} and $(J\ P_1\ \ldots Pn)$ the justification of L at the next higher level of abstraction (i.e., the justification of L before A_{del} was performed).

Lines-To-Remove:=$\{L' | L' \in$ *expansion-segment of* $A_{del}\} - \{L, P_1, \ldots, P_n\}$.
New-Tasks:=$[T]$.
Old-Open-Lines:=$\{L' | L' \in$ *open-lines of* $A_{add}\}$.
Old-Line-Tasks:=$[L' \blacktriangleleft \mathcal{SUPPS}_{L'}$ | L' *in Old-Open-Lines*$]$.
Old-Inst-Tasks:=$[mv|^{Inst}$ | $mv \in$ *Lines-To-Remove and nowhere else in* $\mathcal{PDS}]$.
Old-Exp-Tasks:=
$[L'|^{Exp}$ |($L' \in$ *Lines-To-Remove or* $L' = L$) *and* L' *closed by tactic*$]$.
Tasks-To-Remove:= *Old-Line-Tasks* ∪ *Old-Inst-Tasks* ∪ *Old-Exp-Tasks*.

If A_{del} is deletable with respect to $\vec{A}$ and if Â, $\mathcal{P}$, and $\vec{\mathcal{B}}$S resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result ($\vec{A}$',Â',$\mathcal{P}$',$\vec{\mathcal{B}}$S') of $\Phi^{-1}_{\text{MULTI}}(\vec{A}, \hat{A}, \mathcal{P}, \vec{\mathcal{B}}S, A_{del})$ is defined by:

- $\vec{A}$':=$\vec{A}$ − A_{del}.
- Â':= *New-Tasks* ∪ (Â − *Tasks-To-Remove*).
- $\mathcal{P}$' results from $\mathcal{P}$ by

1. removing the current justification from L and setting $(J\ P_1\ \ldots Pn)$ as the current one, and
2. removing the proof lines in *New-Lines*.

- $\vec{\mathcal{B}}\mathrm{S}$':=$\vec{\mathcal{B}}\mathrm{S}$.

□

Definition 7.24 ($\Phi^{-1}_{\mathrm{MULTI}}$ on CPLANNER or PPLANNER Actions): Let $\vec{A}$ be a sequence of actions and let A_{del} be a **CPLANNER** or a **PPLANNER** action in $\vec{A}$. Let $\vec{\mathcal{B}}\mathrm{S}$ be a sequence of bindings stores, $\mathcal{P}$ a $\mathcal{PDS}$, and $\hat{\mathrm{A}}$ an agenda. Moreover, let $[A_1, \ldots, A_n]$ be the action-sequence of A_{del}.
$(\vec{A}_{rec}, \hat{\mathrm{A}}_{rec}, \mathcal{P}_{rec}, \vec{\mathcal{B}}\mathrm{S}_{rec}) := \vec{\Phi}^{-1}_{\mathrm{MULTI}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, [A_n, \ldots, A_1])$
If A_{del} is deletable with respect to $\vec{A}$ and if $\hat{\mathrm{A}}$, $\mathcal{P}$, and $\vec{\mathcal{B}}\mathrm{S}$ resulted from the introduction of $\vec{A}$ (to some agenda, $\mathcal{PDS}$, and sequence of binding stores), then the result $(\vec{A}', \hat{\mathrm{A}}', \mathcal{P}', \vec{\mathcal{B}}\mathrm{S}')$ of $\Phi^{-1}_{\mathrm{MULTI}}(\vec{A}, \hat{\mathrm{A}}, \mathcal{P}, \vec{\mathcal{B}}\mathrm{S}, A_{del})$ is defined by:

- $\vec{A}$':= $\vec{A}_{rec} - [A_{del}]$.[9]
- $\hat{\mathrm{A}}$':= $\hat{\mathrm{A}}_{rec}$.
- $\mathcal{P}$':= $\mathcal{P}_{rec}$.
- $\vec{\mathcal{B}}\mathrm{S}$':=$\vec{\mathcal{B}}\mathrm{S}_{rec}$.

□

With these definitions at our disposal, we can now describe the **BACKTRACK** algorithm. Figure 7.15 contains a pseudo-code description of **BACKTRACK**. **BACKTRACK** obtains two arguments. First, a pair (S, T), which consists of a **BACKTRACK** strategy S and a task T. Second, the list of all alternative job offers on the control blackboard, when the **BACKTRACK** strategy was invoked. **BACKTRACK** returns no specific output but updates the content of the proof blackboard.

Step 1 applies the computation function of the strategy S to the task T. This returns a sequence of actions that **BACKTRACK** should delete, and **BACKTRACK** binds the algorithm variable $\vec{A}_{del}$ to this action sequence. Moreover, **BACKTRACK** writes a backtrack-start entry with this information to the history.

The steps 2-5 are essentially a while-loop, which is passed through until $\vec{A}_{del}$ is empty. First, Step 2 checks whether $\vec{A}_{del}$ is empty. If this is the case, it creates a success message,[10] writes a backtrack-stop entry with this message to the history, and terminates. Otherwise, step 3 picks the first action from $\vec{A}_{del}$ and stores it in the algorithm variable A_{del}. A_{del} is then either deleted in step 5 or step 4 extends $\vec{A}_{del}$ depending on A_{del}. Step 4 first checks whether A_{del} is deletable with respect to the sequence of actions on the proof blackboard. If this is not the case, then there are actions which depend on A_{del} and step 4 adds these actions, which are computed by the function *dependend-actions*, in front of $\vec{A}_{del}$. If A_{del} is deletable, then step 4 checks next whether it is an action of **PPLANNER** or **CPLANNER** whose action-sequence is not empty. If this holds, then it adds the action sequence of A_{del} in front of $\vec{A}_{del}$. Otherwise, step 5 is reached, which uses $\Phi^{-1}_{\mathrm{MULTI}}$ to delete A_{del} and

[9] When all actions in A_{del} are deleted, then A_{del} remains with an empty action sequence. Here A_{del} itself is deleted from the action sequence.

[10] Note that **BACKTRACK** is not supposed to fail (except of hopefully not occurring programming errors).

Input: (1) a pair (S,T) where S is a **BACKTRACK** strategy and T is a task, (2) the list of all alternative job offers $\mathcal{J}$.

Output: No output, only changes of the blackboards.

```
Algorithm: BACKTRACK($(S,T),\mathcal{J}$)

  1. Compute Actions To Be Deleted
     Let $\vec{A}_{del}$:=apply(compute-del-actions-function($S$),$T$).
     add-backtrack-start-record-to-history($\mathcal{J}$,$\vec{A}_{del}$).

  2. Terminate
     If $\vec{A}_{del}=\emptyset$
        then
           Let $message$:=create-success-message($S,T$).
           add-backtrack-stop-record-to-history($message$).
           Terminate.

  3. Select Action
     Let $A_{del}$:=first($\vec{A}_{del}$).

  4. Extend Actions
     If $A_{del}$ is not deletable wrt. the sequence of actions on PB
        then
           $\vec{A}_{del}$:=dependend-actions($A_{del}$) $\cup$ $\vec{A}_{del}$.
           Goto step 3.

     If $A_{del}$ is CPLANNER or PPLANNER action, whose action-sequence is not
     empty
        then
           $\vec{A}_{del}$:=action-sequence($A_{del}$) $\cup$ $\vec{A}_{del}$.
           Goto step 3.

  5. Delete Action
     PB:=$\Phi^{-1}_{\text{MULTI}}(A_{del},PB)$.
     add-action-del-record($A_{del}$).
     Let $\vec{A}_{del}$:=$\vec{A}_{del}$ $-$ $[A_{del}]$.
     If action-of-terminated-strategy($A_{del}$)
        then
           write-to-memory(get-tasktag($A_{del}$),$\emptyset$).
     Goto step 2.
```

Figure 7.15: The **BACKTRACK** algorithm.

to update the action sequence, the agenda, the $\mathcal{PDS}$, and the sequence of binding stores on the proof blackboard. Moreover, it adds an action-deletion entry to the history and removes A_{del} from $\vec{A}_{del}$.

If the deleted action A_{del} belongs to a terminated **PPLANNER** or **CPLANNER** strategy execution (this is checked by the function *action-of-terminated-strategy*), then a re-invokation of this strategy execution should be enabled again. **BACKTRACK** reactivates the strategy execution by writing an entry to the memory consisting of the task tag of the strategy execution (which is computed by the function *get-tasktag* from the history) and an empty set of demand pointers. From this memory entry the terminated strategy execution can be re-invoked.

Note that **BackTrack** could apply $\Phi^{-1}_{\textsc{Multi}}$ directly to actions of **PPlanner** and **CPlanner** that are not empty (since we did define $\Phi^{-1}_{\textsc{Multi}}$ for such actions in definition 7.24). However, **BackTrack** first successively deletes the action sequence of an action of **PPlanner** and **CPlanner** before it deletes the "empty" **PPlanner** or **CPlanner** action. This guarantees that detailed history information for each deleted action is created (i.e, for each action, which is in the action-sequence of an action of **PPlanner** or **CPlanner** as well as for the **PPlanner** or **CPlanner** action itself).

7.6 Remarks

7.6.1 Representing the Search with Trees

The check for dependency among actions as well as the changes caused by backtracking of an action are complex operations as described in the previous section. The problem is that the $\mathcal{PDS}$, which is the central data structure in the current implementation of ΩMEGA and MULTI, is a complex data structure difficult to maintain. In the ongoing re-implementation of the ΩMEGA system on top of the CORE system [9] we suggest an agenda as the (only) central data structure. Moreover, we suggest additional data structures to considerably simplify the backtracking of actions.

The introduction of an action into a strategic proof plan reduces a task to a set of tasks, which can be empty. The introduced actions and the resulting tasks could be stored in a tree, a so-called *task-action-tree*, whose nodes are labeled with the tasks and whose edges are labeled with the actions.[11] Figure 7.16 depicts such a task-action-tree. The root node of the tree is labeled with the initial task. If this tree is constructed during the strategic proof planning process, then the current agenda consists always of the tasks of the leave nodes of the tree.

With a task-action-tree the dependency among actions could be formulated as follows: An action A_i depends on another action A_j if the path from the root node to A_i contains A_j. The changes caused by the backtracking of an action could also be stated simpler than currently: If a deletable action A is backtracked, then the children tasks of the action A are removed and the parent task is introduced again into the agenda.

7.6.2 Creating Different Kinds of Solution Proof Plans

In section 7.3, we defined three different notions of strategic solution proof plans, namely method-level solution proof plans, instantiated method-level solution proof plans, and full solution proof plans. In order to produce a method-level solution proof plan MULTI can ignore the instantiation tasks and the expansion-tasks; to produce an instantiated method-level solution proof plans MULTI can ignore only the expansion-tasks; to create a full solution proof plan MULTI has to tackle all kinds of tasks.

In three of the case studies (see the subsequent chapters) we are interested in instantiated method-level solution proof plans. The reason for this is that, in general, we separate in ΩMEGA the search for a solution proof plan from the expansion process.[12] In the case study on proof planning permutation group problems (see

[11] Actually, we use multi-edges that connect one parent node with several children nodes.

[12] An exception is when the expansion of a complex step will provide information needed to tackle existing tasks. For instance, when the expansion of a complex step provides further constraints on meta-variables, which helps to solve existing line-tasks.

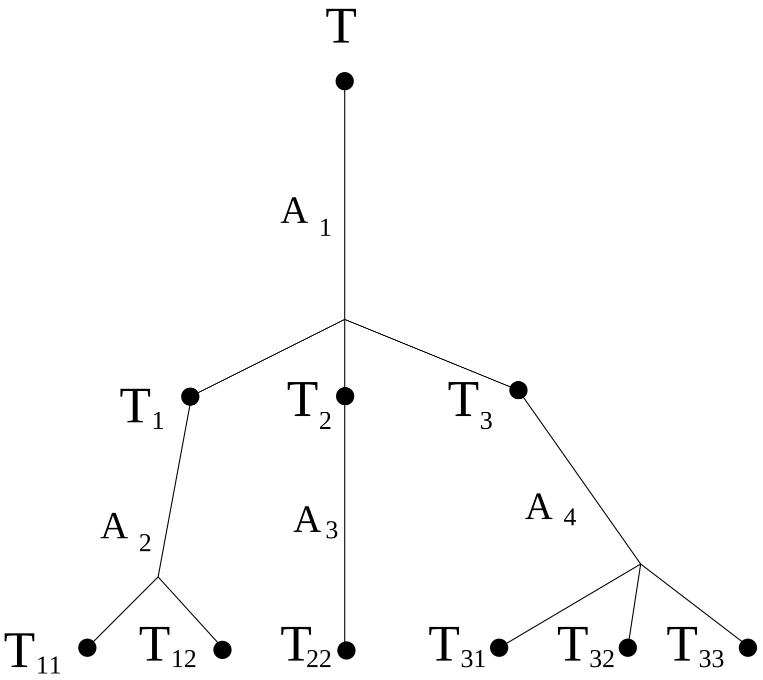


Figure 7.16: A task-action-tree.

section 10.1) we use hierarchical proof planning and expansion to hide proofs of simple subproblems. This allows to come up fast with abstract proof plans for complex problems. Afterwards, the subproblems are opened again and tackled themselves with proof planning.

The simplest possibility to make MULTI search for a particular kind of solution proof plan is to prohibit some strategies. For instance, if there are no strategies of **EXP**, then expansion-tasks will be ignored and MULTI will search for an instantiated method-level solution proof plan. In the case studies it turned out that this approach has the drawback that expansion-tasks are created although they are ignored later on. Therefore, we avoid the creation of not desired expansion-tasks. The user can declare methods or tactics whose applications he wants to be expanded by MULTI as *not-reliable*. MULTI creates expansion-tasks only for such proof lines L whose justification $(J\ P_1 \dots P_N)$ uses a not-reliable method or tactic J.

7.6.3 Cooperation with Constraint Solvers

So far, the only constraint solver connected with MULTI is $\mathcal{C}o\mathcal{SIE}$. MULTI communicates directly with $\mathcal{C}o\mathcal{SIE}$ by interfaces in methods and strategies. When a method action is introduced that contains constraints for $\mathcal{C}o\mathcal{SIE}$, then these constraints are passed to $\mathcal{C}o\mathcal{SIE}$. Moreover, the two strategies InstIfDetermined and ComputeInstFromCS employ $\mathcal{C}o\mathcal{SIE}$ to obtain new bindings. If several constraint solvers should be connected with MULTI, then a direct communication is not sufficient anymore. First, constraints should be passed to all connected constraint solvers for which they are relevant. Second, several constraint solvers should be able to directly exchange results without involving MULTI.

As possible solution we suggest a *constraint solver coordination module*, which handles all communication and which stores all constraints and results. Each constraint solver that should be connected has to register by the coordination module. MULTI passes new constraints to this module. Then, the module asks the connected constraints solvers whether this constraint is relevant for them and passes it to the relevant constraint solvers. The module performs the same distribution, if a constraint solver produces an intermediate result (i.e., when $\mathcal{C}o\mathcal{SIE}$ detects that the instantiation of meta-variable mv is already determined by its current constraints). When MULTI backtracks and deletes some method actions with constraints, then the coordination module has also to organize the deletion of the constraints in the affected constraint solvers and the deletion of intermediate that depend on these constraints.

The module handles and distrubutes also queries of MULTI. MULTI passes queries (e.g., is the instantiation of meta-variable mv already determined?) only to the coordination module. Either the coordination module can answer the query directly (e.g., if an result passed by a connected constraint solver was already a unique instantiation for mv) or it distributes the query to the connected constraint solvers and passes the answer back to MULTI.

7.6.4 Dependencies in Backtracking

When the **BACKTRACK** algorithm removes an action, then it also removes all actions that depend on this action (see section 7.5.7). The notion of dependency for actions used by **BACKTRACK** (see definition 7.17) is strict and therefore **BACKTRACK** may removes more actions than necessary. In particular, the deletion of an **INSTMETA** action causes the deletion of all actions following this action in the current action sequence. We decided for this approach since a more detailed analysis of which following actions actually depend on a new binding is difficult and is still open.

Nevertheless, there are also dependencies between actions that are not covered by the dependency notion in definition 7.17. In particular, there can be various dependencies between actions that involve cooperation with constraint solvers (e.g., $\mathcal{C}o\mathcal{SIE}$). For instance, if the current constraints (e.g., $mv \leq t$ and $mv \geq t$) in $\mathcal{C}o\mathcal{SIE}$ determine the instantiation t for a meta-variable mv, then the strategy InstIfDetermined is applicable with respect to mv and introduces the binding $mv{:=}^b\, t$ into the strategic proof plan. Other actions can rely on this binding. When a method action that contains constraints for $\mathcal{C}o\mathcal{SIE}$ is backtracked, then mv may is not longer determined with respect to the resulting constraint store (e.g., if the constraint $mv \leq t$ is removed). In this case, the action of InstIfDetermined, which binds mv to t, has to be removed. Since this is not a problem of strategies of **INSTMETA** in general but of ComputeInstFromCS in particular, we did not implement such a dependency analysis into the **BACKTRACK** algorithm (i.e., it is not contained in the dependency notion introduced in definition 7.17). Rather we suggest to check such particular dependencies in strategic control rules that cause further backtracking.

The described problematic situation is handled by the strategic control rule `check-det-insts`. `check-det-insts` checks whether the last strategy execution was a **BACKTRACK** step and whether it removed some method actions with constraints for $\mathcal{C}o\mathcal{SIE}$. If this is the case, it checks whether all actions of InstIfDetermined in the current sequence of actions are still valid in the sense that the meta-variables that they bind are still determined in $\mathcal{C}o\mathcal{SIE}$. Then, `check-det-insts` prefers backtracking for each action of InstIfDetermined that is no longer valid.

7.6.5 Failure Information in Execution Messages

When a strategy execution fails, then its algorithm creates a failure message. If possible the algorithm can attach information to a failure message, which can also be used by the control rules. For instance, **PPLANNER** can create and attach information why no applicable action could be found. This functionality affects many single steps in **PPLANNER** and in the procedures **CHOOSEACTION** and **CHOOSEACTIONALL**, which compute and select the next action to be applied. Hence, for the sake of simplicity and clarity, we did not describe this functionality in the algorithms themselves but give an informal description here.

That the procedures **CHOOSEACTION** and **CHOOSEACTIONALL** fail to to provide an action for a line-task T and a method M can be caused by three reasons:

Failed matching of proof lines The $\ominus$ conclusions of M do not match with the task line of T or the blank and $\ominus$ premises of M do not match with the supports of T.

Failed application conditions The evaluation of the application conditions of M can fail with respect to the substitution resulting from a successful matching of the proof lines of M with the task line and the supports of T.

Rejected actions Actions can be rejected by control rules or because they were already applied and then backtracked later on.

These tests are performed successively in **CHOOSEACTION** and **CHOOSEACTIONALL** in this order. Each time such a test fails, the function that performs the test creates an information record. For instance, when the function *eval-appl-conds* finds that the application condition App_c of method M fails with respect to the incomplete action A (which resulted from the successful matching of the proof lines of M with the proof lines of the given task), then *eval-appl-conds* creates the information record $applcondfailure(App_c, M, A)$. **CHOOSEACTION** and **CHOOSEACTIONALL** collect these information records and return them to **PPLANNER**. If there is no applicable action, then **PPLANNER** attaches the set of information records to the created failure message. An example where we make use of such failure information is given in section 8.2.2.

Part III

Case Studies

Introduction to the Case Studies

In the previous chapters we described the architecture and the algorithms of MULTI. In part III of the thesis we shall discuss the case studies we conducted to test the approach. Before we start with the actual description of the case studies, we briefly introduce each case study (without technical details).

The Limit Domain

In chapter 8, we present the application of MULTI to the limit domain. Originally, this domain was tackled with the previous proof planner PLAN (see [172]). The problems we encountered, when tackling the domain with PLAN, gave rise to the development of MULTI as discussed in section 6.1. In this chapter we focus on examples in the limit domain that illustrate the benefits of MULTI and why MULTI can solve problems on which PLAN fails.

The main means to tackle limit problems is the **PPLANNER** strategy SolveInequality. This strategy contains the domain-specific knowledge (i.e., methods and control rules) on how to perform ϵ-δ-proofs. We complement this strategy with two strategies that contain domain-independent methods for the decomposition of complex logical formulas in goals and supports, respectively. The incorporation of the constraint solver $\mathcal{C}o\mathcal{SIE}$ via two **INSTMETA** strategies is also crucial to accomplish ϵ-δ-proofs with MULTI. We integrated a **CPLANNER** strategy to reuse backtracked proof parts. As an alternative to ϵ-δ-proofs we present another **PPLANNER** strategy that solves limit problems by the application of known theorems from ΩMEGA's database.

When discussing this case study, we shall describe how MULTI supports

1. the flexible introduction of instantiations for meta-variables provided by the constraint solver $\mathcal{C}o\mathcal{SIE}$,
2. the flexible cooperation of several strategies driven by interrupts and demands,
3. meta-reasoning on failed proof attempts to guide backtracking or plan modifications (in particular, we shall describe how failures can be exploited to guide the eureka steps case-split introduction and lemma speculation).

The Residue Class Domain

Chapter 9 presents the case study on proof planning for the residue class domain. As opposed to the limit domain, the residue class domain was never tackled with PLAN. We developed several **PPLANNER** strategies as the main strategies to solve residue class problems. They correspond to mathematical proof techniques for tackling the residue class problems. We complement these strategies with two **INSTMETA** strategies, and two **ATP** strategies. The two **INSTMETA** strategies interface two computer algebra systems (namely MAPLE [200] and GAP [93]), a model generator (namely SEM [253]), and a system for theory formation (namely HR [58]) to obtain instantiations for meta-variables. Moreover, we integrated the **PPLANNER** strategies with different backtrack techniques.

We use this case study to illustrate how MULTI supports

1. the modeling of different proof techniques in different strategies, which can produce different proof plans for the same problem,
2. the flexible incorporation of instantiations provided by computer algebra systems, model generators, and systems for theory formation,

3. the integration of different backtrack techniques guided by meta-reasoning,
4. the failure-driven cooperation of strategies,
5. the application of randomization and restart techniques,
6. the flexible cooperation of several strategies.

Permutation Group Domain and Homomorphism Problems

In chapter 10, we shall briefly discuss two further case studies conducted with Multi. In the first case study we apply Multi to solve problems of permutation groups. In the second case study we tackle homomorphism theorems with Multi. We discuss these two case studies since they address hierarchical proof planning with expansion and interactive theorem proving, two issues that are not addressed in the case studies on limit problems and residue class problems.

Chapter 8

The Limit Domain

In this chapter, we present the application of MULTI to the limit domain. Theorems of the limit domain make statements about the limit $\lim\limits_{x \to a} f(x)$ of a function f at a point a, about the limit $limseq\,X$ of a sequence X, about the continuity of a function f at a point a, and about the derivative of a function f at a point a (see section 5.1 for a formal introduction of the limit domain).

The chapter is structured as follows. First, we describe how MULTI creates ϵ-δ-proof plans with the **PPLANNER** strategy SolveInequality and some complementary strategies. Afterwards, we illustrate in section 8.2 how meta-reasoning can exploit failures to guide backtracking and the subsequent proof planning process. In the discussed situations meta-reasoning on the failures is necessary to solve the problems since the failures hold the key to the discovery of a solution proof plan. In section 8.3, we describe how MULTI solves limit problems by the application of known theorems. We conclude this chapter with a discussion of the results of the case study, a discussion of related work, and an evaluation of the realized proof planning approach. An account of all limit problems that MULTI can currently solve is given in Appendix C.

When illustrating the application of MULTI with examples, we try to avoid the tedious details. In particular, we skip the technical details of the constructed strategic proof plans. Rather, we use the $\mathcal{PDS}$ as a means to display and discuss the constructed proof plans. In general, a $\mathcal{PDS}$ is a three-dimensional data structure that can represent (partial) proof attempts at different levels of abstraction (see section 3.2.3). Since the discussed examples exploit no expansion the constructed $\mathcal{PDS}$s consist only of one level of abstraction and are presented in the linearized form described in section 3.1.3.

8.1 ϵ-δ-Proof Plans with MULTI

To accomplish ϵ-δ-proof plans MULTI combines the **PPLANNER** strategies NormalizeLineTask, UnwrapHyp, and SolveInequality and the **INSTMETA** strategies InstIfDetermined and ComputeInstFromCS (see section 6.2.1), which interface $\mathcal{C}o\mathcal{SIE}$. In the following, we illustrate how MULTI employs these strategies with the LIM+ example (introduced in section 5.1) and the first part of exercise 4.1.3 (introduced in section 6.1.1). However, before we elaborate the examples we discuss the employed strategies and their cooperation.

8.1.1 The Strategies and Their Cooperation

The strategy SolveInequality (see Table 6.1 in section 6.2.1) is central for accomplishing ϵ-δ-proofs with Multi. It is applicable to prove line-tasks whose goals are inequalities or whose goals can be reduced to inequalities. A goal is reducible to inequalities if it contains defined terms whose unfolding will result in inequalities, for instance, *lim*, *limseq*, *cont*, and *deriv*. SolveInequality unfolds occurrences of these concepts both in the goal and in the supports of the task. The method for unfolding defined concepts in goals is DefnUnfold-B, whereas DefnUnfold-F unfolds defined concepts in supports.

When faced with an inequality goal, SolveInequality first tries to apply the methods TellCS-B and AskCS-B, which both employ $\mathcal{C}o\mathcal{SIE}$. TellCS-B passes the goal to $\mathcal{C}o\mathcal{SIE}$, whereas AskCS-B asks $\mathcal{C}o\mathcal{SIE}$ whether the goal is entailed by its current constraints. If an inequality is too complex to be handled by $\mathcal{C}o\mathcal{SIE}$, then SolveInequality tries to apply methods that reduce an inequality to simpler inequalities. So, SolveInequality successively produces simpler inequalities, until it reaches inequalities that are accepted by $\mathcal{C}o\mathcal{SIE}$. This approach — handle with $\mathcal{C}o\mathcal{SIE}$ or simplify — is guided by the control rule `prove-inequality` given in Figure 8.1, which is the central control rule in SolveInequality.

```
(control-rule prove-inequality
              (kind methods)
              (IF (and (goal-matches (REL A B))
                       (in REL {<,>,≤,≥})))
              (THEN (prefer (TellCS-B TellCS-F AskCS-B Simplify-B
                             Simplify-F Solve*-B ComplexEstimate-B
                             FactorialEstimate-B SetFocus-B))))
```

Figure 8.1: The control rule `prove-inequality`.

In its IF-part `prove-inequality` checks whether the current goal is an inequality. If this is the case, it prefers the methods TellCS-B, TellCS-F, AskCS-B, Simplify-B, Simplify-F, Solve*-B, ComplexEstimate-B, FactorialEstimate-B, and SetFocus-B in this order. We discussed the methods TellCS-B, TellCS-F, AskCS-B, and ComplexEstimate-B already in section 4.1.4. The method Solve*-B is described in section 5.1. Simplify-B passes the formula of a given goal to the computer algebra system Maple and asks Maple to simplify it. If Maple succeeds, then the given goal is reduced to a new goal with the simplified formula. The analogous method Simplify-F derives a support with a simpler formula from a given support by calling Maple. The method FactorialEstimate-B deals with fractions in inequalities. It reduces a goal of the form $|\frac{t}{t'}| < t''$ to the three subgoals $0 < mv_F$, $mv_F < |t'|$, and $|t| < t'' * mv_F$, where mv_F is a new meta-variable. SetFocus-B highlights a subformula in a support. SolveInequality contains also some further methods whose application is not guided by the control rule `prove-inequality`. We shall introduce and explain these methods as we go along.

SolveInequality comprises the knowledge of how to deal with inequalities and with problems that can be reduced to inequalities. As opposed thereto, the strategies NormalizeLineTask and UnwrapHyp comprise the domain-independent, general knowledge of how to decompose complex formulas with logical connectives and quantifiers. SolveInequality decides once for the decomposition of a complex goal or the unwrapping of a subformula from a complex support. Then, it switches to NormalizeLineTask or UnwrapHyp, which perform all single decomposition steps. This saves SolveInequality from reasoning permanently on the application of methods that decompose single logical connectives and quantifiers such as $\wedge$I-B or $\wedge$E-F.

Technically, the cooperation between SolveInequality and NormalizeLineTask and UnwrapHyp works as follows. For line-tasks whose goals are complex formulas that contain inequality subformulas (e.g., goals that arise from unfolding *lim*, *limseq*, *cont*, or *deriv*) SolveInequality interrupts and places a demand for the strategy NormalizeLineTask on the control blackboard. Guided by this demand, MULTI invokes NormalizeLineTask, which decomposes the complex goal. When re-invoked by MULTI, SolveInequality can tackle the inequalities in the resulting goals. The switch from SolveInequality to UnwrapHyp is driven by missing support inequalities, which are needed for the application of the methods COMPLEXESTIMATE-B and SOLVE*-B. If the other methods preferred by `prove-inequality` fail, then the application of SETFOCUS-B highlights a subformula in an existing support. Afterwards, SolveInequality interrupts and places a demand for the invocation of UnwrapHyp to unwrap the highlighted subformula. When the subformula is unwrapped, SolveInequality can continue with a new support that may enable further steps. The application of SETFOCUS-B (i.e., the selection of the support and the subformula to highlight) is guided by the control rule `choose-unwrap-support` for the supports and parameters choice point. `choose-unwrap-support` analyzes the supports of the task on which the other methods are not applicable. It searches for inequality subformulas in the supports that are similar to the goal of the task. The idea is that similar formulas are likely to unify with the goal such that COMPLEXESTIMATE-B and SOLVE*-B become applicable.

To accomplish ϵ-δ-proofs plans also two **INSTMETA** strategies, namely ComputeInstFromCS and InstIfDetermined, are used that interface the constraint solver $\mathcal{C}o\mathcal{SIE}$. Whereas InstIfDetermined asks $\mathcal{C}o\mathcal{SIE}$ for instantiations of meta-variables that are already determined by the collected constraints, ComputeInstFromCS asks $\mathcal{C}o\mathcal{SIE}$ to compute instantiations for the occurring meta-variables that are consistent with the collected constraints.

The invocation of ComputeInstFromCS is delayed by the strategic control rule `delay-ComputeInstCosie` until all line-tasks are closed. This delay of the computation of instantiations for meta-variables is sensible, since the instantiations should not be computed before all constraints are collected, that is, not before all line-tasks are closed (see discussion in section 6.1.1). However, when the current constraints already determine a meta-variable, then a further delay of the corresponding instantiation is not necessary. Rather, immediate instantiations of determined meta-variables can simplify a problem as we shall see in section 8.1.3.

To enable the flexible instantiation of determined meta-variables SolveInequality cooperates with the strategy InstIfDetermined. Technically, this works as follows. When $\mathcal{C}o\mathcal{SIE}$ signals that a meta-variable is determined, then the control rule `eager-instantiate` in SolveInequality fires. It interrupts SolveInequality and places a demand for InstIfDetermined with respect to the determined meta-variable. After the introduction of a binding for the meta-variable by InstIfDetermined MULTI re-invokes SolveInequality.

8.1.2 The LIM+ Example

In this section, we shall discuss the application of MULTI to the LIM+ problem with the strategies described in the previous section. The LIM+ problem states that the limit of the sum of two functions f and g equals the sum of their limits. That is, the problem states that

$$
\begin{array}{rl}
 & LIM+\text{: } \lim_{x\to a}(f(x)+g(x)) = l_f + l_g \\
\text{follows from} & Lim_f\text{: } \lim_{x\to a} f(x) = l_f \\
\text{and} & Lim_g\text{: } \lim_{x\to a} g(x) = l_g .
\end{array}
$$

Figure 8.2 and Figure 8.3 show the interesting parts, i.e., the parts created by SolveInequality, of the resulting $\mathcal{PDS}$. We indicate the contributions of NormalizeLineTask and UnwrapHyp by justifications in the $\mathcal{PDS}$ such as (UnwrapHyp L_3) (in line L_{49}) and (NormalizeLineTask L_8 L_{12}) (in line L_1), which abbreviate the proof segments created by these strategies. The complete $\mathcal{PDS}$ is given in appendix B. Note that we describe the proof planning process in progress. Hence, we introduce meta-variables, when they arise. When there is a binding for a meta-variable during the proof planning process, then the proof lines created after the introduction of the binding use the instantiation of the meta-variable in order to clarify the following computations.

Lim_f.	Lim_f	$\vdash \lim_{x\to a} f(x) = l_f$	(Hyp)
Lim_g.	Lim_g	$\vdash \lim_{x\to a} g(x) = l_g$	(Hyp)
L_2.	Lim_f	$\vdash \forall \epsilon_1\centerdot(0 < \epsilon_1 \Rightarrow \exists \delta_1\centerdot(0 < \delta_1 \wedge$ $\forall x_1\centerdot(\lvert x_1 - a\rvert < \delta_1 \wedge \lvert x_1 - a\rvert > 0$ $\Rightarrow \lvert f(x_1) - l_f\rvert < \epsilon_1)))$	(DefnUnfold-F Lim_f)
L_3.	Lim_g	$\vdash \forall \epsilon_2\centerdot(0 < \epsilon_2 \Rightarrow \exists \delta_2\centerdot(0 < \delta_2 \wedge$ $\forall x_2\centerdot(\lvert x_2 - a\rvert < \delta_2 \wedge \lvert x_2 - a\rvert > 0$ $\Rightarrow \lvert g(x_2) - l_g\rvert < \epsilon_2)))$	(DefnUnfold-F Lim_g)
L_{21}.	L_{21}	$\vdash 0 < c_{\delta_1} \wedge \forall x_1\centerdot(\lvert x_1 - a\rvert < c_{\delta_1} \wedge \lvert x_1 - a\rvert > 0$ $\Rightarrow \lvert f(x_1) - l_f\rvert < mv_{\epsilon_1})$	(Hyp)
L_{42}.	L_{42}	$\vdash 0 < c_{\delta_2} \wedge \forall x_2\centerdot(\lvert x_2 - a\rvert < c_{\delta_2} \wedge \lvert x_2 - a\rvert > 0$ $\Rightarrow \lvert g(x_2) - l_g\rvert < mv_{\epsilon_2})$	(Hyp)
L_{11}.	L_{11}	$\vdash \lvert c_x - a\rvert > 0 \wedge \lvert c_x - a\rvert < mv_\delta$	(Hyp)
L_5.	L_5	$\vdash 0 < c_\epsilon$	(Hyp)
L_{52}.	$\mathcal{H}_2$	$\vdash mv_{x_2} \doteq c_x$	(TellCS-B)
L_{53}.	$\mathcal{H}_2$	$\vdash mv_{\epsilon_2} \leq \frac{1}{2} * c_\epsilon$	(TellCS-B)
L_{49}.	$\mathcal{H}_2$	$\vdash \lvert g(mv_{x_2}) - l_g\rvert < mv_{\epsilon_2}$	(UnwrapHyp L_3)
L_{48}.	$\mathcal{H}_2$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	(Solve*-B L_{49} L_{52} L_{53})
L_{37}.	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	(UnwrapHyp L_3 L_{48} L_{39} L_{50} L_{51})
L_{31}.	$\mathcal{H}_1$	$\vdash \lvert 1\rvert \leq mv$	(TellCS-B)
L_{32}.	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$	(TellCS-B)
L_{33}.	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{c_\epsilon}{2}$	(Simplify-B L_{37})
L_{34}.	$\mathcal{H}_1$	$\vdash 0 < mv$	(TellCS-B)
L_{35}.	$\mathcal{H}_1$	$\vdash mv_{x_1} \doteq c_x$	(TellCS-B)
L_{28}.	$\mathcal{H}_1$	$\vdash \lvert f(mv_{x_1}) - l_f\rvert < mv_{\epsilon_1}$	(UnwrapHyp L_2)
L_{27}.	$\mathcal{H}_1$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	(ComplexEstimate-B L_{28} L_{31} L_{32} L_{33} L_{34} L_{35})
L_{16}.	$\mathcal{H}_3$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	(UnwrapHyp L_2 L_{27} L_{18} L_{29} L_{30})
L_{12}.	$\mathcal{H}_3$	$\vdash \lvert(f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$	(Simplify-B L_{16})
L_8.	$\mathcal{H}_4$	$\vdash 0 < mv_\delta$	(TellCS-B)
L_1.	Lim_f, Lim_g	$\vdash \forall \epsilon\centerdot(0 < \epsilon \Rightarrow \exists \delta\centerdot(0 < \delta \wedge$ $\forall x\centerdot(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < \epsilon)))$	(NormalizeLineTask L_8 L_{12})
$LIM+$.	Lim_f, Lim_g	$\vdash \lim_{x\to a}(f(x) + g(x)) = l_f + l_g$	(DefnUnfold-B L_1)

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$
$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

Figure 8.2: ϵ-δ-proof for LIM+ (part I).

The proof planning process starts with the invocation of SolveInequality on the initial task $LIM+ \blacktriangleleft \{Lim_f, Lim_g\}$. SolveInequality first unfolds the occurrences of lim. Afterwards, it switches to NormalizeLineTask, which decomposes the resulting complex goal in line L_1 into the goals $\lvert(f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$ in L_{12}

and $0 < mv_\delta$ in L_8 where c_ϵ and c_x are constants introduced for the universally quantified variables ϵ and x in L_1 and mv_δ is a meta-variable introduced for the existentially quantified variable δ.

Both new goals are inequalities and SolveInequality tackles them guided by the control rule prove-inequality. It closes $0 < mv_\delta$ directly by an application of TellCS-B, which passes the formula to $\mathcal{C}o\mathcal{SIE}$. $|(f(c_x)+g(c_x))-(l_f+l_g)| < c_\epsilon$ is not accepted by $\mathcal{C}o\mathcal{SIE}$ and therefore TellCS-B is not applicable. SolveInequality simplifies this goal to $|((f(c_x)+g(c_x))-l_f)-l_g| < c_\epsilon$ in line L_{16} but then fails to solve this goal with the given supports. choose-unwrap-support detects the subformula $|f(x_1)-l_f| < \epsilon_1$ of L_2 as a promising support and guides the application of the method SetFocus-B to highlight the subformula. This triggers the interruption of SolveInequality and the invocation of UnwrapHyp for this subformula. The application of UnwrapHyp yields the new support $|f(mv_{x_1})-l_f| < mv_{\epsilon_1}$ in line L_{28}, but also the three new goals $0 < mv_{\epsilon_1}$ in line L_{18}, $|mv_{x_1}-a| < c_{\delta_1}$ in L_{29}, and $|mv_{x_1}-a| > 0$ in L_{30}. Here UnwrapHyp introduces the constant c_{δ_1} for the existentially quantified variable δ_1 and the meta-variables mv_{ϵ_1} and mv_{x_1} for the universally quantified variables ϵ_1 and x_1 in L_2.

When SolveInequality is re-invoked, it can apply ComplexEstimate-B to the goal $|((f(c_x)+g(c_x))-l_f)-l_g| < c_\epsilon$ and the new support $|f(mv_{x_1})-l_f| < mv_{\epsilon_1}$. This results in the five new goals $|1| \leq mv$ in L_{31}, $mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$ in L_{32}, $|g(c_x)-l_g| < \frac{c_\epsilon}{2}$ in L_{33}, $0 < mv$ in L_{34}, and $mv_{x_1}\dot{=}c_x$ in L_{35}. Except L_{33} all goals are closed by applications of TellCS-B, which pass the respective formulas as constraints to $\mathcal{C}o\mathcal{SIE}$. Since $mv_{x_1}\dot{=}c_x$ determines mv_{x_1} in $\mathcal{C}o\mathcal{SIE}$ the control rule eager-instantiate fires and interrupts SolveInequality. Its demand causes Multi to invoke InstIfDetermined on the instantiation-task of mv_{x_1}. InstIfDetermined introduces the binding $mv_{x_1}{:=}^b c_x$ into the strategic proof plan.

The re-invoked SolveInequality simplifies $|g(c_x)-l_g| < \frac{c_\epsilon}{2}$ to $|g(c_x)-l_g| < \frac{1}{2}*c_\epsilon$ in L_{37} but then fails on this goal with the existing supports. choose-unwrap-support detects the subformula $|g(x_2)-l_g| < \epsilon_2$ of L_3 as a promising support and guides the corresponding application of the method SetFocus-B to highlight this subformula. Afterwards, SolveInequality interrupts and Multi switches to UnwrapHyp, which unwraps the subformula and yields the new support $|g(mv_{x_2})-l_g| < mv_{\epsilon_2}$ in line L_{49}. The unwrapping yields also the three new goals $0 < mv_{\epsilon_2}$ in line L_{39}, $|mv_{x_2}-a| < c_{\delta_2}$ in L_{50}, and $|mv_{x_2}-a| > 0$ in L_{51}. UnwrapHyp introduces the constant c_{δ_2} for the existentially quantified variable δ_2 and the meta-variables mv_{ϵ_2} and mv_{x_2} for the universally quantified variables ϵ_2 and x_2 in L_3.

When re-invoked, SolveInequality applies Solve*-B to the goal $|g(c_x)-l_g| < \frac{1}{2}*c_\epsilon$ and the new support $|g(mv_{x_2})-l_g| < mv_{\epsilon_2}$. This results in the new goals $mv_{x_2}\dot{=}c_x$ in L_{52} and $mv_{\epsilon_2} \leq \frac{1}{2}*c_\epsilon$ in L_{53}, which SolveInequality closes by TellCS-B. $mv_{x_2}\dot{=}c_x$ determines the meta-variable mv_{x_2} in $\mathcal{C}o\mathcal{SIE}$. Thus, the control rule eager-instantiate suggests a switch from SolveInequality to InstIfDetermined, which introduces the binding $mv_{x_2}{:=}^b c_x$ into the strategic proof plan.

Afterwards, SolveInequality has to deal with the remaining goals L_{18}, L_{29}, L_{30}, and L_{39}, L_{50}, L_{51}, which resulted from the applications of the UnwrapHyp strategy. Figure 8.3 gives the $\mathcal{PDS}$ segment created by SolveInequality for these goals. It closes L_{18} and L_{39} directly by TellCS-B. The inequalities in the other goals cannot be passed to $\mathcal{C}o\mathcal{SIE}$ directly because TellCS-B is not applicable to them. Instead, SolveInequality applies Solve*-B to these goals with supports that stem from the decomposition of the initial goal by NormalizeLineTask. The applications of Solve*-B result in inequality goals, which SolveInequality closes either with TellCS-B or AskCS-B.

L_{18}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_1}$	(TELLCS-B)
L_{39}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_2}$	(TELLCS-B)
L_{11}.	L_{11}	$\vdash \lvert c_x - a\rvert > 0 \wedge \lvert c_x - a\rvert < mv_\delta$	(Hyp)
L_{14}.	L_{11}	$\vdash \lvert c_x - a\rvert > 0$	($\wedge$E-F L_{11})
L_{13}.	L_{11}	$\vdash \lvert c_x - a\rvert < mv_\delta$	($\wedge$E-F L_{11})
L_{61}.	$\mathcal{H}_1$	$\vdash 0 \leq 0$	(ASKCS-B)
L_{59}.	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TELLCS-B)
L_{57}.	$\mathcal{H}_2$	$\vdash 0 \leq 0$	(ASKCS-B)
L_{55}.	$\mathcal{H}_2$	$\vdash mv_\delta \leq c_{\delta_2}$	(TELLCS-B)
L_{29}.	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1}$	(SOLVE*-B L_{13} L_{59})
L_{30}.	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert > 0$	(SOLVE*-B L_{14} L_{61})
L_{50}.	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2}$	(SOLVE*-B L_{13} L_{55})
L_{51}.	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert > 0$	(SOLVE*-B L_{14} L_{57})

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$
$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

Figure 8.3: ϵ-δ-proof for LIM+ (part II).

After closing all line-tasks, SolveInequality terminates. Next, MULTI invokes ComputeInstFromCS on the instantiation-tasks and $\mathcal{CoSIE}$ provides instantiations for the meta-variables that are consistent with the collected constraints (see Figure 5.1 in section 5.1). ComputeInstFromCS inserts these instantiations as the bindings

$$mv{:=}^b 1,\ mv_{\epsilon_1}{:=}^b \tfrac{c_\epsilon}{2},\ mv_{\epsilon_2}{:=}^b \tfrac{c_\epsilon}{2},\ \text{and } mv_\delta{:=}^b min(c_{\delta_1}, c_{\delta_2})$$

into the strategic proof plan.

8.1.3 Eager Instantiation

We discussed already in section 6.1.1 that PLAN fails to solve some limit problems that require the eager instantiation of meta-variables. In the following, we shall see how MULTI solves those problems since it performs eager instantiation guided by the control rule `eager-instantiate`.

We illustrate MULTI's eager meta-variable instantiation with the first part of exercise 4.1.3 in the analysis textbook [12], which states that

$$Thm\text{: } \lim_{x\to 0} f(x+c) = l \text{ follows from } Ass\text{: } \lim_{x_1\to c} f(x_1) = l,$$

Figure 8.4 and Figure 8.5 show the $\mathcal{PDS}$ segments created by SolveInequality for this problem. As in the previous section, we indicate and abbreviate the proof parts generated by NormalizeLineTask and UnwrapHyp by justifications in the $\mathcal{PDS}$.

When invoked on the initial task $Thm \blacktriangleleft \{Ass\}$, SolveInequality unfolds the occurrences of lim in the goal and the supports and then switches to NormalizeLineTask, which decomposes the resulting complex goal. This results in the two goals $0 < mv_\delta$ in L_7 and $|f(c_x{+}c){-}l| < c_\epsilon$ in L_{11} where c_ϵ and c_x are constants introduced for the universally quantified variables ϵ and x in L_1 and mv_δ is a meta-variable introduced for the existentially quantified variable δ.

SolveInequality closes $0 < mv_\delta$ by TELLCS-B but fails to tackle $|f(c_x{+}c){-}l| < c_\epsilon$ with the current supports. A promising support is the subformula $|f(x_1) - l| < \epsilon_1$ of L_2. Thus, after highlighting the subformula with SETFOCUS-B, SolveInequality switches to UnwrapHyp. The application of UnwrapHyp yields the new support $|f(mv_{x_1}) - l| < mv_{\epsilon_1}$ in L_{26} and the new goals $0 < mv_{\epsilon_1}$ in L_{16}, $|mv_{x_1} - c| < c_{\delta_1}$ in L_{27}, and $|mv_{x_1} - c| > 0$ in L_{28}. UnwrapHyp introduces the constant c_{δ_1} for the

$Ass.$	Ass	$\vdash \lim_{x_1 \to c} f(x_1) = l$	(Hyp)
$L_2.$	Ass	$\vdash \forall \epsilon_1 \blacksquare (0 < \epsilon_1 \Rightarrow \exists \delta_1 \blacksquare (0 < \delta_1 \wedge$ $\forall x_1 \blacksquare (\lvert x_1 - c\rvert < \delta_1 \wedge \lvert x_1 - c\rvert > 0$ $\Rightarrow \lvert f(x_1) - l\rvert < \epsilon_1)))$	(DEFNUNFOLD-F Ass)
$L_{19}.$	L_{19}	$\vdash 0 < c_{\delta_1} \wedge \forall x_1 \blacksquare (\lvert x_1 - c\rvert < c_{\delta_1} \wedge \lvert x_1 - c\rvert > 0$ $\Rightarrow \lvert f(x_1) - l\rvert < mv_{\epsilon_1})$	(Hyp)
$L_4.$	L_4	$\vdash 0 < c_\epsilon$	(Hyp)
$L_{29}.$	$\mathcal{H}_1$	$\vdash mv_{x_1} \doteq c_x + c$	(TELLCS-B)
$L_{30}.$	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq c_\epsilon$	(TELLCS-B)
$L_{26}.$	$\mathcal{H}_1$	$\vdash \lvert f(mv_{x_1}) - l\rvert < mv_{\epsilon_1}$	(UnwrapHyp L_2)
$L_{25}.$	$\mathcal{H}_1$	$\vdash \lvert f(c_x + c) - l\rvert < c_\epsilon$	(SOLVE*-B L_{26} L_{29} L_{30})
$L_{11}.$	$\mathcal{H}_2$	$\vdash \lvert f(c_x + c) - l\rvert < c_\epsilon$	(UnwrapHyp L_2 L_{25} L_{16} L_{27} L_{28})
$L_7.$	Ass, L_4	$\vdash 0 < mv_\delta$	(TELLCS-B)
$L_1.$	Ass	$\vdash \forall \epsilon \blacksquare (0 < \epsilon \Rightarrow \exists \delta \blacksquare (0 < \delta \wedge$ $\forall x \blacksquare (\lvert x - 0\rvert < \delta \wedge \lvert x - 0\rvert > 0$ $\Rightarrow \lvert f(x + c) - l\rvert < \epsilon)))$	(NormalizeLineTask L_7 L_{11})
$Thm.$	Ass	$\vdash \lim_{x \to 0} f(x + c) = l$	(DEFNUNFOLD-B L_1)
$\mathcal{H}_1 = \{Ass, L_4, L_{10}, L_{19}\},\ \mathcal{H}_2 = \{Ass, L_4, L_{10}\}$			

Figure 8.4: ϵ-δ-proof for first part of exercise 4.1.3 (part I).

existentially quantified variable δ_1 and the meta-variables mv_{ϵ_1} and mv_{x_1} for the universally quantified variables ϵ_1 and x_1 in L_2.

When re-invoked, SolveInequality applies SOLVE*-B to $\lvert f(c_x + c) - l\rvert < c_\epsilon$ and the new support $\lvert f(mv_{x_1}) - l\rvert < mv_{\epsilon_1}$. This results in the new goals $mv_{x_1} \doteq c_x + c$ in L_{29} and $mv_{\epsilon_1} \leq c_\epsilon$ in L_{30}, which SolveInequality both closes by TELLCS-B. Since $mv_{x_1} \doteq c_x + c$ determines the meta-variable mv_{x_1} in $\mathcal{C}o\mathcal{SIE}$, SolveInequality switches to InstIfDetermined, which introduces the binding $mv_{x_1} :=^b c_x + c$ into the strategic proof plan.

$L_{10}.$	L_{10}	$\vdash \lvert c_x - 0\rvert > 0 \wedge \lvert c_x - 0\rvert < mv_\delta$	(Hyp)
$L_{13}.$	L_{10}	$\vdash \lvert c_x - 0\rvert > 0$	($\wedge$E-F L_{10})
$L_{12}.$	L_{10}	$\vdash \lvert c_x - 0\rvert < mv_\delta$	($\wedge$E-F L_{10})
$L_{36}.$	L_{10}	$\vdash \lvert c_x\rvert > 0$	(SIMPLIFY-F L_{13})
$L_{32}.$	L_{10}	$\vdash \lvert c_x\rvert < mv_\delta$	(SIMPLIFY-F L_{12})
$L_{34}.$	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TELLCS-B)
$L_{31}.$	$\mathcal{H}_1$	$\vdash \lvert c_x\rvert < c_{\delta_1}$	(SOLVE*-B L_{32} L_{34})
$L_{35}.$	$\mathcal{H}_1$	$\vdash \lvert c_x\rvert > 0$	(WEAKEN-B L_{36})
$L_{27}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - c\rvert < c_{\delta_1}$	(SIMPLIFY-B L_{31})
$L_{28}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - c\rvert > 0$	(SIMPLIFY-B L_{35})
$L_{16}.$	$\mathcal{H}_2$	$\vdash 0 < mv_{\epsilon_1}$	(TELLCS-B)
$\mathcal{H}_1 = \{Ass, L_4, L_{10}, L_{19}\},\ \mathcal{H}_2 = \{Ass, L_4, L_{10}\}$			

Figure 8.5: ϵ-δ-proof for first part of exercise 4.1.3 (part II).

Afterwards, SolveInequality has to deal with the remaining goals L_{16}, L_{27}, and L_{28}, which resulted from the application of UnwrapHyp. Figure 8.5 gives the $\mathcal{PDS}$ segment created by SolveInequality for these goals. It closes L_{16} by TELLCS-B. The goals in L_{27} and L_{28} become $\lvert(c_x + c) - c\rvert < c_{\delta_1}$ and $\lvert(c_x + c) - c\rvert > 0$ with respect to the binding $mv_{x_1} :=^b c_x + c$ in the strategic proof plan. Applications of SIMPLIFY-B reduce these two goals to the $\lvert c_x\rvert < c_{\delta_1}$ in L_{31} and $\lvert c_x\rvert > 0$ in L_{35}. SolveInequality closes these new goals with the supports $\lvert c_x\rvert > 0$ and $\lvert c_x\rvert < mv_\delta$ that are derived from L_{10}, which was introduced during the application of NormalizeLineTask.

$\mathcal{C}o\mathcal{SIE}$ has the final constraint store depicted in Figure 8.6. It computes instantiations for the meta-variables that are consistent with these constraints. ComputeInstFromCS inserts these instantiations as the bindings $mv_\delta :=^b c_{\delta_1}$ and $mv_{\epsilon_1} :=^b c_\epsilon$

into the strategic proof plan.

$$\boxed{\begin{array}{lllll} mv_{x_1} = c_x + c & & & & \\ 0 & < & c_{\delta_1} & < & +\infty \\ 0 & < & c_{\epsilon} & < & +\infty \\ 0 & < & mv_{\epsilon_1} & \leq & c_{\epsilon} \\ 0 & < & mv_{\delta} & \leq & c_{\delta_1} \end{array}}$$

Figure 8.6: The final constraint store of $\mathcal{CoSIE}$ for the first part of exercise 4.1.3.

Responsible for the success of SolveInequality on L_{27} and L_{28} is the eager introduction of the binding $mv_{x_1} :=^b c_x + c$. This binding changes the formulas of L_{27} and L_{28} and so SIMPLIFY-B becomes applicable.[1]

Another problem from the limit domain that requires eager meta-variable instantiation is exercise 4.1.12 in [12], which states that

$$Thm\text{: } \lim_{x \to 0} f(a * x) = l \text{ follows from } Ass\text{: } \lim_{x_1 \to 0} f(x_1) = l \text{ for } a > 0.$$

First, MULTI reduces the initial goal $\lim_{x \to 0} f(a*x) = l$ to $|f(a*c_x) - l| < c_\epsilon$. Then, it unwraps the support $|f(mv_{x_1}) - l| < mv_{\epsilon_1}$. The application of SOLVE*-B to this goal and this support results in the goal $mv_{x_1} \doteq a*c_x$, which is passed to $\mathcal{CoSIE}$. Since this formula determines mv_{x_1} the binding $mv_{x_1} :=^b a * c_x$ is introduced into the strategic proof plan. The remaining goals $|mv_{x_1} - 0| < c_{\delta_1}$ and $|mv_{x_1} - 0| > 0$ that result from the unwrapping of the support become $|a * c_x| < c_{\delta_1}$ and $|a * c_x| > 0$ with respect to this binding. They are then solved by applications of COMPLEXESTIMATE-B with the supports $|c_x| > 0$ and $|c_x| < mv_\delta$.[2] See also section 8.2.2 for further examples that require eager meta-variable instantiation.

8.2 Failure Reasoning in the Limit Domain

In this section, we shall discuss three types of situations whose solution requires meta-reasoning on failures. In two situations the failures can be exploited to guide the introduction of case-splits and the speculation of lemmas, two eureka steps whose necessity is difficult to spot and whose introduction is difficult to guide in general. In the third situation we guide backtracking by meta-reasoning on desirable but blocked strategies. All three types of situations have in common that failures in the proof planning process can be productively used and hold the key to discover a solution proof plan.

[1] PLAN, which does not allows for eager meta-variable instantiation, would fail on the goals L_{27} and L_{28} since it cannot close $|mv_{x_1} - c| < c_{\delta_1}$ and $|mv_{x_1} - c| > 0$ from $|c_x| < mv_\delta$ and $|c_x| > 0$ derivable from L_{10}.

[2] PLAN would fail on these goals since without eager meta-variable instantiation it cannot apply COMPLEXESTIMATE-B to solve $|mv_{x_1}| < c_{\delta_1}$ and $|mv_{x_1}| > 0$ with $|c_x| > 0$ and $|c_x| < mv_\delta$, respectively. Rather, it would apply SOLVE*-B to these goals and supports. This results in the subgoal $mv_{x_1} \doteq c_x$, which $\mathcal{CoSIE}$ rejects since it is not consistent with the already collected constraint $mv_{x_1} \doteq a * c_x$. Thus, TELLCS-B is not applicable and PLAN fails.

8.2.1 Guiding Case-Splits

A well-known technique from mathematics to deal with complex problems is to split the problem into cases and to solve the cases separately.[3] But how should the eureka step case-split be controlled? That is, when should MULTI decide for a case-split and which cases should it consider? We found a type of situations in which the need for a case-split and its construction can be spoted by failure reasoning.

As example consider the Cont-If-Deriv problem. This problem states that a function f is continuous at point a if it has a derivative f' at point a. That is,

$$Thm\text{: } cont(f,a) \text{ follows from } Ass\text{: } deriv(f,a) = f'.$$

We give the $\mathcal{PDS}$ segment created by SolveInequality before the failure occurs in Figure 8.7. As in the previous sections we abbreviate the proof parts generated by NormalizeLineTask and UnwrapHyp by strategic justifications in the $\mathcal{PDS}$.

As usual, SolveInequality unfolds the defined concepts and then switches to NormalizeLineTask for the decomposition of the complex goal. The resulting main goal is $|f(c_x) - f(a)| < c_\epsilon$. SolveInequality fails to tackle this goal with the current supports. Since the control rule `choose-unwrap-support` detects the subformula $|\frac{f(x_1)-f(a)}{x_1-a} - f'| < \epsilon_1$ in L_3 as a promising support SolveInequality switches to UnwrapHyp whose application yields the new support $|\frac{f(mv_{x_1})-f(a)}{mv_{x_1}-x} - f'| < mv_{\epsilon_1}$ in line L_{25} and the three new goals $0 < mv_{\epsilon_1}$ in L_{18}, $|mv_{x_1} - a| < c_{\delta_1}$ in L_{26}, and $|mv_{x_1} - a| > 0$ in L_{27}. With the new support SolveInequality closes the main goal $|f(c_x) - f(a)| < c_\epsilon$ in several steps as described in Figure 8.7 (in between SolveInequality interrupts once and switches to InstIfDetermined to introduce the binding $mv_{x_1} :=^b c_x$). Then, it tackles the new goals from the application of UnwrapHyp (see the region between the dashed lines in Figure 8.7). It succeeds to solve L_{18} and L_{26} but fails to solve L_{27} whose formula becomes $|c_x - a| > 0$ with respect to the binding $mv_{x_1} :=^b c_x$ meanwhile introduced.

MULTI succeeded to solve the goal $|f(c_x) - f(a)| < c_\epsilon$ with the derived support $|\frac{f(mv_{x_1})-f(a)}{mv_{x_1}-x} - f'| < mv_{\epsilon_1}$. However, it failed to prove $|c_x - a| > 0$, one of the conditions of the support $|\frac{f(mv_{x_1})-f(a)}{mv_{x_1}-x} - f'| < mv_{\epsilon_1}$. The partial success, i.e., the solution of the initial goal, gives rise to consider to patch the proof attempt by introducing a case-split $|c_x - a| > 0 \vee \neg(|c_x - a| > 0)$ on the failing condition.

In general, the failure and its solution follow this pattern: there is a goal G, which MULTI can solve with a support G' that has some conditions *Conds*. When MULTI uses G', then it introduces the conditions *Conds* as new goals. Afterwards, it fails to prove some of these new goals. We call such a goal a *failing condition*, whereas we call the initial goal G the *main goal*. The failure "failing condition while main goal is solved" can be productively used by introducing a case-split on the failing condition. Then, the main goal G has to be proved several times under different case-split hypotheses.

We shall elaborate this idea with our example. If SolveInequality fails to prove a condition of a support that was used to prove the main goal, then a strategic control rule triggers the backtracking of the unwrapping and the use of the support. In our example, this control rule guides the backtracking of the application of UnwrapHyp and all actions that depend on it such that the resulting proof plan consists only of the unfolding of the defined concepts and the application of NormalizeLineTask. In particular, L_{12} becomes open again. When MULTI re-invokes SolveInequality, then a

[3] SCHOENFELD mentions this technique as a frequently used heuristic: "Decompose the domain of the problem and work on it case by case." ([209] p. 109)

$Ass.$	Ass	$\vdash deriv(f,a) = f'$	(Hyp)
$L_2.$	Ass	$\vdash \lim\limits_{x_1 \to a} \frac{f(x_1)-f(a)}{x_1-a} = f'$	(DefnUnfold-F Ass)
$L_3.$	Ass	$\vdash \forall \epsilon_1{\blacksquare}(0 < \epsilon_1 \Rightarrow \exists \delta_1{\blacksquare}(0 < \delta_1 \wedge$ $\forall x_1{\blacksquare}(\lvert x_1 - a\rvert < \delta_1 \wedge \lvert x_1 - a\rvert > 0$ $\Rightarrow \lvert \frac{f(x_1)-f(a)}{x_1-a} - f'\rvert < \epsilon_1)))$	(DefnUnfold-F L_2)
$L_{15}.$	L_{15}	$\vdash 0 < c_{\delta_1} \wedge \forall x_1{\blacksquare}(\lvert x_1 - a\rvert < c_{\delta_1} \wedge \lvert x_1 - a\rvert > 0$ $\Rightarrow \lvert \frac{f(x_1)-f(a)}{x_1-a} - f'\rvert < mv_{\epsilon_1})$	(Hyp)
$L_{11}.$	L_{11}	$\vdash \lvert c_x - a\rvert < mv_\delta$	(Hyp)
$L_7.$	L_7	$\vdash 0 < c_\epsilon$	(Hyp)
$L_{27}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert > 0$	$(Open)$
$L_{44}.$	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TellCS-B)
$L_{26}.$	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1}$	(Solve*-B L_{11} L_{44})
$L_{18}.$	$\mathcal{H}_2$	$\vdash 0 < mv_{\epsilon_1}$	(TellCS-B)
$L_{42}.$	$\mathcal{H}_1$	$\vdash 0 < \frac{\frac{c_\epsilon}{2}}{2}$	(AskCS-B)
$L_{37}.$	$\mathcal{H}_1$	$\vdash \lvert f'\rvert \leq mv'$	(TellCS-B)
$L_{38}.$	$\mathcal{H}_1$	$\vdash mv_\delta \leq \frac{\frac{c_\epsilon}{2}}{2*mv'}$	(TellCS-B)
$L_{39}.$	$\mathcal{H}_1$	$\vdash \lvert 0\rvert < \frac{\frac{c_\epsilon}{2}}{2}$	(Simplify-B L_{42})
$L_{40}.$	$\mathcal{H}_1$	$\vdash 0 < mv'$	(TellCS-B)
$L_{36}.$	$\mathcal{H}_1$	$\vdash mv_\delta \leq mv$	(TellCS-B)
$L_{28}.$	$\mathcal{H}_1$	$\vdash \lvert x - a\rvert \leq mv$	(Solve*-B L_{11} L_{36})
$L_{29}.$	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$	(TellCS-B)
$L_{30}.$	$\mathcal{H}_1$	$\vdash \lvert f' * c_x - f' * a\rvert < \frac{c_\epsilon}{2}$	(ComplexEstimate-B L_{11} L_{37} L_{38} L_{39} L_{40})
$L_{31}.$	$\mathcal{H}_1$	$\vdash 0 < mv$	(TellCS-B)
$L_{32}.$	$\mathcal{H}_1$	$\vdash mv_{x_1} \doteq c_x$	(TellCS-B)
$L_{25}.$	$\mathcal{H}_1$	$\vdash \lvert \frac{f(mv_{x_1})-f(a)}{mv_{x_1}-x} - f'\rvert < mv_{\epsilon_1}$	(UnwrapHyp L_3)
$L_{24}.$	$\mathcal{H}_1$	$\vdash \lvert f(c_x) - f(a)\rvert < c_\epsilon$	(ComplexEstimate-B L_{25} L_{28} L_{29} L_{30} L_{31} L_{32})
$L_{12}.$	$\mathcal{H}_2$	$\vdash \lvert f(c_x) - f(a)\rvert < c_\epsilon$	(UnwrapHyp L_3 L_{24} L_{18} L_{26} L_{27})
$L_9.$	Ass, L_7	$\vdash 0 < mv_\delta$	(TellCS-B)
$L_1.$	Ass	$\vdash \forall \epsilon{\blacksquare}(0 < \epsilon \Rightarrow \exists \delta{\blacksquare}(0 < \delta \wedge$ $\forall x{\blacksquare}(\lvert x - a\rvert < \delta$ $\Rightarrow \lvert f(x) - f(a)\rvert < \epsilon)))$	(NormalizeLineTask L_9 L_{12})
$Thm.$	Ass	$\vdash cont(f,a)$	(DefnUnfold-B L_1)
$\mathcal{H}_1 = \{Ass, L_7, L_{11}, L_{15}\},\ \mathcal{H}_2 = \{Ass, L_7, L_{11}\}$			

Figure 8.7: ϵ-δ-proof for CONT-IF-DERIV (part I).

control rule in SolveInequality fires that checks whether the last step was backtracking triggered by a failing condition. This control rule then suggests the application of the method CaseSplit-B on the re-opened main goal L_{12} with respect to the failing condition $|c_x - a| > 0$ and its negation $\neg(|c_x - a| > 0)$. This results in the $\mathcal{PDS}$ in Figure 8.8.

Afterwards, SolveInequality has to prove $|f(c_x) - f(a)| < c_\epsilon$ twice: once in L_{47} with hypothesis $|c_x - a| > 0$ and once in L_{49} with hypothesis $\neg(|c_x - a| > 0)$. To tackle L_{47} SolveInequality does not again perform proof search from the scratch. Rather, triggered by a control rule, it switches to the **CPlanner** strategy TaskDirectedAnalogy, which transfers the backtracked proof segment to a proof plan for L_{47}. The failing condition $|c_x - a| > 0$ now follows from the hypothesis of the case. The second case in L_{49} is solved differently by SolveInequality. First, it simplifies the hypothesis $\neg(|c_x - a| > 0)$ to $c_x \doteq a$. Afterwards, it applies this equation with =Subst-B to $|f(c_x) - f(a)| < c_\epsilon$ in L_{49}. The resulting goal $|f(a) - f(a)| < c_\epsilon$ can be simplified with Simplify-B to $0 < c_\epsilon$, which follows from L_7.

Cont-If-Lim=f and Lim-If-Both-Sides-Lim are other problems that require this

$Ass.$	Ass	$\vdash deriv(f,a) = f'$	(Hyp)
$L_2.$	Ass	$\vdash \lim\limits_{x_1 \to a} \frac{f(x_1)-f(a)}{x_1-a} = f'$	(DefnUnfold-F Ass)
$L_3.$	Ass	$\vdash \forall \epsilon_1 \bullet (0 < \epsilon_1 \Rightarrow \exists \delta_1 \bullet (0 < \delta_1 \wedge$ $\forall x_1 \bullet (\lvert x_1 - a\rvert < \delta_1 \wedge \lvert x_1 - a\rvert > 0$ $\Rightarrow \lvert \frac{f(x_1)-f(a)}{x_1-a} - f'\rvert < \epsilon_1)))$	(DefnUnfold-F L_2)
$L_{11}.$	L_{11}	$\vdash \lvert c_x - a\rvert < mv_\delta$	(Hyp)
$L_7.$	L_7	$\vdash 0 < c_\epsilon$	(Hyp)
$L_{45}.$	L_{45}	$\vdash \lvert c_x - a\rvert > 0 \vee \neg(\lvert c_x - a\rvert > 0)$	(TertiumNonDatur)
$L_{48}.$	L_{48}	$\vdash \neg(\lvert c_x - a\rvert > 0)$	(Hyp)
$L_{49}.$	$\mathcal{H}_4$	$\vdash \lvert f(c_x) - f(a)\rvert < c_\epsilon$	$(Open)$
$L_{46}.$	L_{46}	$\vdash \lvert c_x - a\rvert > 0$	(Hyp)
$L_{47}.$	$\mathcal{H}_3$	$\vdash \lvert f(c_x) - f(a)\rvert < c_\epsilon$	$(Open)$
$L_{12}.$	$\mathcal{H}_2$	$\vdash \lvert f(c_x) - f(a)\rvert < c_\epsilon$	(CaseSplit-B L_{45} L_{47} L_{49})
$L_9.$	Ass, L_7	$\vdash 0 < mv_\delta$	(TellCS-B)
$L_1.$	Ass	$\vdash \forall \epsilon \bullet (0 < \epsilon \Rightarrow \exists \delta \bullet (0 < \delta \wedge$ $\forall x \bullet (\lvert x - a\rvert < \delta$ $\Rightarrow \lvert f(x) - f(a)\rvert < \epsilon)))$	(NormalizeLineTask L_9 L_{12})
$Thm.$	Ass	$\vdash cont(f,a)$	(DefnUnfold-B L_1)
$\mathcal{H}_3 = \{Ass, L_7, L_{11}, L_{45}, L_{46}\},\ \mathcal{H}_2 = \{Ass, L_7, L_{11}\}$			
$\mathcal{H}_4 = \{Ass, L_7, L_{11}, L_{45}, L_{46}\}$			

Figure 8.8: ϵ-δ-proof for CONT-IF-DERIV (part II).

kind of failure reasoning. Cont-If-Lim=f states that a function f is continuous at point a if the limit at point a is $f(a)$. The unfolding of the definitions and the application of NormalizeLineTask result in the main goal $|f(c_x) - f(a)| < c_\epsilon$ that can be solved by unwrapping $|f(mv_{x_1}) - f(a)| < mv_{\epsilon_1}$ from the assumption. However, the subgoal $|c_x - a| > 0$ that is created by UnwrapHyp cannot be solved. This failing condition triggers the same case-split and the same solution of the resulting two cases as in the Cont-If-Deriv problem. The Lim-If-Both-Sides-Lim problem states that a function f has a limit l at point a, if both the right-hand and the left-hand limit of f at a are l.[4] Unfolding of the definitions and the application of NormalizeLineTask result in the main goal $|f(c_x) - l| < c_\epsilon$. A support to solve the main goal can be unwrapped either from the right-hand limit assumption or from the left-hand limit assumption. However, in both cases the application of UnwrapHyp yields an condition that cannot be closed. For instance, when UnwrapHyp unwraps the right-hand limit assumption, then there is the failing condition $c_x - a > 0$. This failing condition triggers the case-split into the cases $c_x - a > 0$ and $\neg(c_x - a > 0)$ for the main goal $|f(c_x) - l| < c_\epsilon$. Whereas the first case can be solved by unwrapping the right-hand limit assumption, the second case requires to unwrap the left-hand limit.

8.2.2 Lemma Speculation

It is common mathematical practice to speculate lemmas during a proof attempt and to prove the lemmas separately. Since technically arbitrary formulas can be introduced, lemma speculation introduces an infinite branching point into the search space that is difficult to control in automated theorem proving. We found a type of situations in which suitable (and necessary) lemmas can be speculated by failure reasoning.

[4] *Right-hand and left-hand limit* are defined as follows:

$$limR_{(\nu\nu)\nu\nu o} \equiv \begin{array}{l} \lambda f_{\nu\nu} \bullet \lambda a_\nu \bullet \lambda l_\nu \bullet \forall \epsilon_\nu \bullet (0 < \epsilon \Rightarrow \\ \quad \exists \delta_\nu \bullet (0 < \delta \wedge \forall x_\nu \bullet (x - a > 0 \wedge x - a < \delta \Rightarrow |f(x) - l| < \epsilon))) \end{array}$$

$$limL_{(\nu\nu)\nu\nu o} \equiv \begin{array}{l} \lambda f_{\nu\nu} \bullet \lambda a_\nu \bullet \lambda l_\nu \bullet \forall \epsilon_\nu \bullet (0 < \epsilon \Rightarrow \\ \quad \exists \delta_\nu \bullet (0 < \delta \wedge \forall x_\nu \bullet (a - x > 0 \wedge a - x < \delta \Rightarrow |f(x) - l| < \epsilon))) \end{array}$$

As example consider the second part of exercise 4.1.3 from the analysis textbook [12]. This problem states that

$$Thm\text{: } \lim_{x_1 \to c} f(x_1) = l \text{ follows from } Ass\text{: } \lim_{x \to 0} f(x + c) = l.$$

Figure 8.9 depicts the $\mathcal{PDS}$ segment created by SolveInequality until the failure occurs. As in the previous section, we indicate and abbreviate the proof parts generated by NormalizeLineTask and UnwrapHyp by strategic justifications.

$Ass.$	Ass	$\vdash \lim_{x \to 0} f(x + c) = l$	(Hyp)
$L_2.$	Ass	$\vdash \forall \epsilon_\blacksquare (0 < \epsilon \Rightarrow \exists \delta_\blacksquare (0 < \delta \wedge$ $\forall x_\blacksquare (\lvert x - 0\rvert < \delta \wedge \lvert x - 0\rvert > 0$ $\Rightarrow \lvert f(x + c) - l\rvert < \epsilon)))$	(DefnUnfold-F Ass)
$L_{19}.$	L_{19}	$\vdash 0 < c_\delta \wedge \forall x_\blacksquare (\lvert x - 0\rvert < c_\delta \wedge \lvert x - 0\rvert > 0$ $\Rightarrow \lvert f(x + c) - l\rvert < mv_\epsilon)$	(Hyp)
$L_4.$	L_4	$\vdash 0 < c_{\epsilon_1}$	(Hyp)
$L_{10}.$	L_{10}	$\vdash \lvert c_{x_1} - c\rvert > 0 \wedge \lvert c_{x_1} - c\rvert < mv_\delta$	(Hyp)
$L_{27}.$	$\mathcal{H}_1$	$\vdash \lvert mv_x - c\rvert < c_{\delta_1}$	$(Open)$
$L_{28}.$	$\mathcal{H}_1$	$\vdash \lvert mv_x - c\rvert > 0$	$(Open)$
$L_{16}.$	$\mathcal{H}_2$	$\vdash 0 < mv_\epsilon$	$(Open)$
$L_{26}.$	$\mathcal{H}_1$	$\vdash \lvert f(mv_x + c) - l\rvert < mv_\epsilon$	(UnwrapHyp L_2)
$L_{25}.$	$\mathcal{H}_1$	$\vdash \lvert f(c_{x_1}) - l\rvert < c_{\epsilon_1}$	$(Open)$
$L_{11}.$	$\mathcal{H}_2$	$\vdash \lvert f(c_{x_1}) - l\rvert < c_{\epsilon_1}$	(UnwrapHyp L_2 L_{25} L_{16} L_{27} L_{28})
$L_7.$	Ass, L_4	$\vdash 0 < mv_{\delta_1}$	(TellCS-B)
$L_1.$	Ass	$\vdash \forall \epsilon_{1\blacksquare} (0 < \epsilon_1 \Rightarrow \exists \delta_{1\blacksquare} (0 < \delta_1 \wedge$ $\forall x_{1\blacksquare} (\lvert x_1 - c\rvert < \delta_1 \wedge \lvert x_1 - c\rvert > 0$ $\Rightarrow \lvert f(x_1) - l\rvert < \epsilon_1)))$	(NormalizeLineTask L_7 L_{11})
$Thm.$	Ass	$\vdash \lim_{x_1 \to c} f(x_1) = l$	(DefnUnfold-B L_1)

$\mathcal{H}_1 = \{Ass, L_4, L_{10}, L_{19}\}$, $\mathcal{H}_2 = \{Ass, L_4, L_{10}\}$

Figure 8.9: ϵ-δ-proof for second part of exercise 4.1.3 (part I).

SolveInequality unfolds the defined concepts and then switches to NormalizeLineTask, which decomposes the complex goal. This results in the goal $|f(c_{x_1}) - l| < c_{\epsilon_1}$ in L_{11}, which SolveInequality cannot tackle with the given supports. Hence, it switches to UnwrapHyp in order to decompose the subformula $|f(x + c) - l| < \epsilon$ in L_2. The application of UnwrapHyp yields the new support $|f(mv_x + c) - l| < mv_\epsilon$ in line L_{26} and the three additional goals $0 < mv_\epsilon$ in L_{16}, $|mv_x - 0| < c_\delta$ in L_{27}, and $|mv_x - 0| > 0$ in L_{28}.

Next, SolveInequality should apply Solve*-B to tackle $|f(c_{x_1}) - l| < c_{\epsilon_1}$ with the new support $|f(mv_x + c) - l| < mv_\epsilon$. However, this fails since the application condition *unify* of Solve*-B is not satisfied, that is, the unification algorithm fails to unify $|f(mv_x + c) - l|$ and $|f(c_{x_1}) - l|$. Since no other method is applicable and there is also no further promising subformula to unwrap, Multi would backtrack next. The analysis that $|f(mv_x + c) - l|$ and $|f(c_{x_1}) - l|$ are quite similar and that the unification is blocked only because of the residue $mv_x + c = c_{x_1}$ give rise to consider to patch the proof attempt by speculating the residue $mv_x + c = c_{x_1}$ as lemma.

In general, the failure and its solution follow this pattern: A method tests in its application conditions for a unifier or a matching of two terms t and t'. The unification or matching of t and t' fails because of some residues. If these residues look promising to be provable in the current context, then they are speculated as lemmas. The lemmas are used to rewrite the initial terms such that afterwards the unification or matching succeeds and the method becomes applicable.

The question is, when is a residue promising to be provable in the current context? In the limit domain, we exploit the constraint solver $\mathcal{C}o\mathcal{SIE}$ to decide whether residues are promising lemmas. Whereas the employed unification and matching are decidable procedures that depend on no domain-specific knowledge, $\mathcal{C}o\mathcal{SIE}$ employs domain knowledge of inequalities and equations over the field of real numbers. To exploit this domain knowledge as well as the context information passed to $\mathcal{C}o\mathcal{SIE}$ so far we query $\mathcal{C}o\mathcal{SIE}$ whether it accepts the residues before we speculate them as lemmas. In this way, we combine the domain-independent unification and matching with the domain knowledge contained in $\mathcal{C}o\mathcal{SIE}$.[5]

Technically, the described productive use of failing unifications and matchings for lemma speculation is encoded in the control rule `choose-equation-residues` in SolveInequality. `choose-equation-residues` analyzes the residues of blocked unifications and matchings and queries $\mathcal{C}o\mathcal{SIE}$ whether it accepts the residues. If this is the case, `choose-equation-residues` fires and suggests the application of the method =Subst*-B. This method rewrites a goal by simultaneously applying a set of equations. The equations are given as parameters to =Subst*-B and become new goals, i.e., are speculated as lemmas.

We shall elaborate this approach with our example. When SolveInequality fails to tackle $|f(c_{x_1})-l| < c_{\epsilon_1}$ with the new support $|f(mv_x+c)-l| < mv_\epsilon$, then MULTI creates the failure record

$$applcondfailure(\mathit{unify}(|f(mv_x+c)-l|, |f(c_{x_1})-l|), \text{SOLVE*-B}, A')$$

for the method SOLVE*-B. This failure record states that the evaluation of the application condition *unify* of the method SOLVE*-B failed for $|f(mv_x+c)-l|$ and $|f(c_{x_1})-l|$. The analysis of the failure record by `choose-equation-residues` yields the residue $mv_x + c = c_{x_1}$, which is accepted by $\mathcal{C}o\mathcal{SIE}$. Hence, the control rule `choose-equation-residues` fires and guides the application of =Subst*-B with $mv_x + c \doteq c_{x_1}$ as new lemma.

L_{30}.	$\mathcal{H}_1$	$\vdash mv_x + c \doteq c_{x_1}$	(TELLCS-B)
L_{31}.	$\mathcal{H}_1$	$\vdash mv_\epsilon \leq c_{\epsilon_1}$	(TELLCS-B)
L_{26}.	$\mathcal{H}_1$	$\vdash \lvert f(mv_x + c) - l\rvert < mv_\epsilon$	(UnwrapHyp L_2)
L_{29}.	$\mathcal{H}_1$	$\vdash \lvert f(mv_x + c) - l\rvert < c_{\epsilon_1}$	(SOLVE*-B L_{26} L_{31})
L_{25}.	$\mathcal{H}_1$	$\vdash \lvert f(c_{x_1}) - l\rvert < c_{\epsilon_1}$	(=Subst*-B L_{29} L_{30})

Figure 8.10: ϵ-δ-proof for second part of exercise 4.1.3 (part II).

Figure 8.10 displays the application of =Subst*-B and the following $\mathcal{PDS}$ segment computed by SolveInequality for our example. The application of =Subst*-B to the goal $|f(c_{x_1})-l| < c_{\epsilon_1}$ in L_{25} results in the new goals $|f(mv_x+c)-l| < c_{\epsilon_1}$ in L_{29} and $mv_x + c \doteq c_{x_1}$ in L_{30}. SolveInequality closes $mv_x + c \doteq c_{x_1}$ with TELLCS-B, which passes the constraint to $\mathcal{C}o\mathcal{SIE}$. $|f(mv_x+c)-l| < c_{\epsilon_1}$ is closed by SOLVE*-B with respect to the support $|f(mv_x + c) - l| < mv_\epsilon$ in L_{26}. This is now possible since the unification became unblocked. The resulting goal in L_{31} is closed by TELLCS-B.

$\mathcal{C}o\mathcal{SIE}$ derives $mv_x \doteq c_{x_1} - c$ from the given formula $mv_x + c \doteq c_{x_1}$. This determines mv_x, so that SolveInequality switches to InstIfDetermined, which introduces the binding $mv_x :=^b c_{x_1} - c$ into the strategic proof plan. With respect to this binding the remaining goals in L_{27} and L_{28} become $|(c_{x_1} - c) - 0| < c_\delta$ and $|(c_{x-1} - c) - 0| > 0$.

[5] An alternative to this combination is theory unification, which incorporates domain-specific equations into the unification procedures. However, the decidability of theory unification is difficult to determine and depends on the concrete set of domain equations (e.g., see [25]). We prefer decidable unification and matching procedure in order to avoid undecidable application conditions whose evaluation can block the complete proof planning process.

Applications of SIMPLIFY-B reduce these goals to $|c_{x_1} - c| < c_\delta$ and $|c_{x-1} - c| > 0$, which SolveInequality closes with supports derived from line L_{10}.

Another problem from the limit domain, which requires a similar speculation of lemmas is the reverse of exercise 4.1.12 from [12], which states that

$$Thm: \lim_{x_1 \to 0} f(x_1) = l \text{ follows from } Ass: \lim_{x \to 0} f(a * x) = l \text{ and } a > 0.$$

Unfolding of lim and normalization result in the goal $|f(c_{x_1}) - l| < c_{\epsilon_1}$. The Unwrapping of the assumption yields $|f(a * mv_x) - l| < mv_\epsilon$. The application of SOLVE*-B with respect to these two terms is blocked since the unification has the residue $a * mv_x = c_{x_1}$. Since $\mathcal{C}o\mathcal{SIE}$ accepts the constraint $a * mv_x \doteq c_{x_1}$ SolveInequality can unblock the unification and can apply SOLVE*-B. $\mathcal{C}o\mathcal{SIE}$ yields $\frac{c_{x1}}{a}$ as instantiation for mv_x.[6]

8.2.3 Goal-Directed Backtracking

Goal-directed reasoning selects and applies steps in order to achieve some given goals. That is, a step is either chosen since it directly achieves some of the current goals or since its effects enable some other desirable steps that are likely to help to achieve given goals. Typically, in search procedures backtracking is not a goal-directed operation in its own right but only a necessary operation to traverse the search space. MULTI provides the freedom to backtrack any actions in the proof plan under construction. This allows for *goal-directed backtracking*, that is, backtracking that is not just part of the traversal of the search space but that aims to work towards the current goals by enabling desirable steps. In this section, we shall discuss a type of situation in which goal-directed backtracking is suggested by meta-reasoning on a highly desirable but blocked strategy.

As example problem consider the problem LIM-DIV-1-X, which states that

$$Thm: \lim_{x \to c} \tfrac{1}{x} = \tfrac{1}{c} \text{ for } c > 0.$$

Figure 8.11 depicts the $\mathcal{PDS}$ that is created for this problem before the highly desirable but blocked strategy occurs.

The unfolding of the defined symbol lim and the normalization of the resulting complex goal results in the two goals $0 < mv_\delta$ in L_6 and $|\frac{1}{c_x} - \frac{1}{c}| < c_\epsilon$ in L_9. SolveInequality closes the first goal by an application of TELLCS-B whereas it simplifies the second goal to $|\frac{c - c_x}{c_x * c}| < c_\epsilon$ in L_{12}. An application of FACTORIALESTIMATE-B to this goal results in the three goals $0 < mv_f$ in L_{13}, $|c_x * c| > mv_f$ in L_{14}, and $|c - c_x| < mv_f * c_\epsilon$ in L_{15}. SolveInequality closes these three goals with TELLCS-B.

Since then all line-tasks are closed $\mathcal{C}o\mathcal{SIE}$ is supposed to provide instantiations for the meta-variables mv_δ and mv_f that are consistent with the collected constraints. That is, the strategy ComputeInstFromCS, which asks $\mathcal{C}o\mathcal{SIE}$ to compute the instantiations, becomes a highly desirable strategy. However, $\mathcal{C}o\mathcal{SIE}$ fails to compute instantiations in this situation and ComputeInstFromCS does not succeed. What is the problem? So far, $\mathcal{C}o\mathcal{SIE}$ did collect the constraints

[6] This is another example that needs eager meta-variable instantiation. Since $a * mv_x \doteq c_{x_1}$ determines mv_x, the binding $mv_x :=^b \frac{c_{x1}}{a}$ is introduced into the proof plan. The unwrapping of the support also yields the two goals $|mv_x - 0| < c_\delta$ and $|mv_x - 0| > 0$, which are simplified with respect to the binding to $|\frac{c_{x1}}{a}| < c_\delta$ and $|\frac{c_{x1}}{a}| > 0$. Whereas MULTI can solve these two goals from the supports $|c_{x_1}| > 0 \wedge |c_{x_1}| < mv_\delta$ by applications of COMPLEXESTIMATE-B, PLAN fails to prove the goals without the eager instantiation.

$Ass.$	Ass	$\vdash 0 < c$	(Hyp)
$L_8.$	L_8	$\vdash \lvert c_x - c\rvert < mv_\delta \wedge \lvert c_x - c\rvert > 0$	(Hyp)
$L_4.$	L_7	$\vdash 0 < c_\epsilon$	(Hyp)
$L_{10}.$	L_8	$\vdash \lvert c_x - c\rvert < mv_\delta$	($\wedge$E-F L_8)
$L_{11}.$	L_8	$\vdash \lvert c_x - c\rvert > 0$	($\wedge$E-F L_8)
$L_{13}.$	$\mathcal{H}_1$	$\vdash 0 < mv_f$	(TELLCS-B)
$L_{14}.$	$\mathcal{H}_1$	$\vdash \lvert c_x * c\rvert > mv_f$	(TELLCS-B)
$L_{15}.$	$\mathcal{H}_1$	$\vdash \lvert c - c_x\rvert < mv_f * c_\epsilon$	(TELLCS-B)
$L_{12}.$	$\mathcal{H}_1$	$\vdash \lvert \frac{c-c_x}{c_x * c}\rvert < c_\epsilon$	(FACTORIALESTIMATE-B L_{13} L_{14} L_{15})
$L_9.$	$\mathcal{H}_1$	$\vdash \lvert \frac{1}{c_x} - \frac{1}{c}\rvert < c_\epsilon$	(SIMPLIFY-B L_{12})
$L_6.$	Ass, L_7	$\vdash 0 < mv_\delta$	(TELLCS-B)
$L_1.$	Ass	$\vdash \forall \epsilon_{\blacksquare}(0 < \epsilon \Rightarrow \exists \delta_{\blacksquare}(0 < \delta \wedge$ $\forall x_{\blacksquare}(\lvert x - c\rvert < \delta \wedge \lvert x - c\rvert > 0$ $\Rightarrow \lvert \frac{1}{x} - \frac{1}{c}\rvert < \epsilon)))$	(NormalizeLineTask L_6 L_9)
$Thm.$	Ass	$\vdash \lim\limits_{x \to c} \frac{1}{x} = \frac{1}{c}$	(DEFNUNFOLD-B L_1)

$\mathcal{H}_1 = \{Ass, L_4, L_8\}$

Figure 8.11: ϵ-δ-proof for LIM-DIV-1-X before failure.

$$\frac{|c_x - c|}{c_\epsilon} < mv_f,\ 0 < mv_f,\ mv_f < |c_x * c|,\ 0 < mv_\delta,\ 0 < c, \text{ and } 0 < c_\epsilon.$$

The critical constraints are the constraints on mv_f that state that $\frac{|c_x - c|}{c_\epsilon}$ has to be less than mv_f, which has to be less than $|c_x * c|$. These constraints are consistent, but a solution for mv_f exists only, if $\frac{|c_x - c|}{c_\epsilon} < |c_x * c|$ holds. This, however, does not follow from the constraints collected so far. In particular, the constraints collected so far are not sufficient for an ϵ-δ-proof since they do not establish a connection between the ϵ and the δ.

A possibility to overcome this problem is to refine the existing constraints in order to obtain an extended set of refined constraints for which a solution exists. That is, applications of TELLCS-B have to be backtracked in a goal-directed manner in order to enable further refinement of some constraints.

We encoded the described idea in the strategic control rule `backtrack-to-unblock-cosie`. When all line-tasks are closed, but ComputeInstFromCS is not applicable since $\mathcal{CoSIE}$ fails to compute instantiations, then this control rule analyzes the constraints passed to $\mathcal{CoSIE}$ by TELLCS-B. It triggers the backtracking of actions of TELLCS-B that pass complex inequalities to $\mathcal{CoSIE}$ that can be further refined.[7] When SolveInequality tackles the re-opened proof lines, it cannot close them again with TELLCS-B but has to refine them. Afterwards, it can pass the refined goals to $\mathcal{CoSIE}$.

We shall elaborate this idea with our example. Triggered by the strategic control rule `backtrack-to-unblock-cosie` MULTI backtracks the application of TELLCS-B that closes L_{15}. SolveInequality reduces the re-opened goal L_{15} with COMPLEXESTIMATE-B. Afterwards, it passes the resulting inequality goals by applications of TELLCS-B to $\mathcal{CoSIE}$. Since $\mathcal{CoSIE}$ also fails on this extended constraint set MULTI backtracks the application of TELLCS-B that closes L_{14}. Again, SolveInequality reduces the re-opened goal with COMPLEXESTIMATE-B and passes the resulting inequalities to $\mathcal{CoSIE}$. The new $\mathcal{PDS}$ segments for L_{14} and L_{15} are shown in Figure 8.12.

This results in the following constraint store:

[7] Currently, the critical constraints are chosen by some heuristics encoded in `backtrack-to-unblock-cosie`. It would be more convenient, if $\mathcal{CoSIE}$ would directly point out what the critical constraints are. However, this kind of information is not provided by the current $\mathcal{CoSIE}$ system.

L_{10}.	L_8	$\vdash \lvert c_x - c \rvert < mv_\delta$	($\wedge$E-F L_8)
L_{11}.	L_8	$\vdash \lvert c_x - c \rvert > 0$	($\wedge$E-F L_8)
L_{22}.	$\mathcal{H}_1$	$\vdash 0 < mv'$	(TellCS-B)
L_{23}.	$\mathcal{H}_1$	$\vdash \lvert c \rvert < mv'$	(TellCS-B)
L_{24}.	$\mathcal{H}_1$	$\vdash \lvert c * c \rvert \geq mv_f * 2$	(TellCS-B)
L_{25}.	$\mathcal{H}_1$	$\vdash mv_\delta \leq \frac{mv_f}{mv'}$	(TellCS-B)
L_{14}.	$\mathcal{H}_1$	$\vdash \lvert c_x * c \rvert > mv_f$	(ComplexEstimate-B L_{10} L_{22} L_{23} L_{24} L_{25})
L_{17}.	$\mathcal{H}_1$	$\vdash \lvert -1 \rvert \leq mv$	(TellCS-B)
L_{18}.	$\mathcal{H}_1$	$\vdash mv_\delta \leq \frac{c_\epsilon * mv_f}{2 * mv}$	(TellCS-B)
L_{19}.	$\mathcal{H}_1$	$\vdash \lvert 0 \rvert < \frac{c_\epsilon * mv_f}{2}$	(TellCS-B)
L_{20}.	$\mathcal{H}_1$	$\vdash 0 < mv$	(TellCS-B)
L_{15}.	$\mathcal{H}_1$	$\vdash \lvert c - c_x \rvert < mv_f * c_\epsilon$	(ComplexEstimate-B L_{10} L_{17} L_{18} L_{19} L_{20})

Figure 8.12: Extended ϵ-δ-proof for LIM-DIV-1-X.

$$\begin{array}{llll}
c_\epsilon > 0 & c > 0 & mv_f \geq mv' * mv_\delta & mv' > c \\
mv_f > 0 & mv > 1 & \frac{c_\epsilon * mv_f}{2} > 0 & mv_\delta > 0 \\
mv_\delta \leq \frac{c_\epsilon * mv_f}{2 * mv} & mv_f * 2 \leq c^2 & &
\end{array}$$

Bindings that are consistent with these constraints are: $mv :=^b 2$, $mv' :=^b c + 1$, $mv_f :=^b \frac{c^2}{2}$, and $mv_\delta :=^b min(\frac{c_\epsilon * c^2}{8}, \frac{c^2}{2*(c+1)})$. Unfortunately, the solution of the above constraint system is not in the scope of the current $\mathcal{CoSIE}$ system. That is, $\mathcal{CoSIE}$ fails to provide instantiations although a solution that is consistent with all constraints exists and establishes a connection between the ϵ and the δ of our ϵ-δ-proof.[8] Since `backtrack-to-unblock-cosie` detects no further inequality goals that probably can be further refined Multi terminates without bindings for the meta-variables. Despite the successful failure analysis that triggered goal-directed backtracking, the problem cannot be solved completely because of drawbacks of the current $\mathcal{CoSIE}$ system.

All problems of the limit domain that result in absolute values of fractions that are tackled with FactorialEstimate-B need the described failure reasoning. For instance, exercises 4.1.10(a) – (d) in [12]:

$$\lim_{x \to 2} \frac{1}{1-x} = -1, \ \lim_{x \to 1} \frac{x}{x+1} = \frac{1}{2}, \ \lim_{x \to 0} \frac{x^2}{|x|} = 0, \ \lim_{x \to 1} \frac{x^2 - x + 1}{x+1} = \frac{1}{2},$$

and problems on the derivative of functions such as theorem 6.1.3(a) and (b) in [12]:

$$\begin{array}{l}
deriv(f, a) = f' \Rightarrow deriv(\alpha * f, a) = \alpha * f', \\
deriv(f, a) = f' \wedge deriv(g, a) = g' \Rightarrow deriv(f + g, a) = f' + g'.
\end{array}$$

Note that the current $\mathcal{CoSIE}$ system fails for all these problems to compute suitable instantiations.

8.3 Applying Theorems

Sometimes, different sections of mathematical textbooks introduce different ways to tackle the same problem based on different theory contexts. A typical structure

[8] The reason for $\mathcal{CoSIE}$ failing to find this solution is the mutual dependency of the variables mv_f and mv_δ. mv_f occurs in an upper bound of mv_δ, and in turn mv_δ occurs in a lower bound of mv_f. The search procedure of the current $\mathcal{CoSIE}$ system is not complete in a sense that it can not resolve all dependencies of this kind.

is, for instance, to prove first some basic theorems with a basic technique and to use these theorems afterwards to prove further problems. In the textbook [12] both the chapter on the limit of sequences (chapter three) and the chapter on the limit of functions (chapter four) are structured in this way. First ϵ-δ-proofs are used as a basic technique to tackle limit problems (section 1 of chapter three and chapter four, respectively), then these theorems are used to prove more problems (section 2 of chapter three and chapter four, respectively). In the previous sections of this chapter, we discussed how MULTI solves limit problems with the basic ϵ–δ technique. In the following, we shall discuss how MULTI can solve limit problems by using known theorems and how it combines the application of theorems with the ϵ–δ technique.

For the application of known theorems we encoded an extra strategy, ReduceToSpecial. The central method in ReduceToSpecial is APPLYASS-B, which applies theorems from ΩMEGA's theory database. APPLYASS-B can apply a theorem to a goal, if the conclusion of the theorem matches the goal. The application of the method results in the premises of the theorem to be the new open goals. Moreover, ReduceToSpecial contains several methods that close particular goals that are often created by the application of theorems (e.g., the methods INTI-B and REALI-B that close goals of the form $n \in \mathbb{Z}$ or $r \in \mathbb{R}$ where n and r are concrete numbers). ReduceToSpecial also contains the TELLCS-B method, which is used to pass equations with meta-variables to $\mathcal{C}o\mathcal{SIE}$.

ReduceToSpecial creates shorter and more abstract proofs for some problems that MULTI could also solve with ϵ-δ-proofs. Moreover, the strategy extends the solvability horizon of MULTI for the limit domain since the combination of ReduceToSpecial and SolveInequality can solve problems on which SolveInequality alone fails. We exemplify ReduceToSpecial with the two problems $\lim\limits_{x\to 1}(x+1)*(2*x+3) = 10$ (exercise 4.2.1(a) in [12]) and $\lim\limits_{x\to 0} sin(x) = 0$ (example 4.2.8(b) in [12]) that demonstrate both aspects of ReduceToSpecial.

The proof of $\lim\limits_{x\to 1}(x+1)*(2*x+3) = 10$ with ReduceToSpecial relies on the following theorems in ΩMEGA's database:

$$\begin{aligned}
&LIM+: && \forall f. \forall g. \forall c. \forall l. \forall l_f. \forall l_g. (\lim_{x\to c} f(x) = l_f \wedge \lim_{x\to c} g(x) = l_g \\
& && \qquad \wedge\, l_f + l_g \doteq l) \Rightarrow \lim_{x\to c} f(x) + g(x) = l \\
&LIM*: && \forall f. \forall g. \forall c. \forall l. \forall l_f. \forall l_g. (\lim_{x\to c} f(x) = l_f \wedge \lim_{x\to c} g(x) = l_g \\
& && \qquad \wedge\, l_f * l_g \doteq l) \Rightarrow \lim_{x\to c} f(x) * g(x) = l \\
&LIMV: && \forall c. \forall l. l \doteq c \Rightarrow \lim_{x\to c} x = l \\
&LIMC: && \forall a. \forall c. \forall l. l \doteq a \Rightarrow \lim_{x\to c} a = l
\end{aligned}$$

Figure 8.13 displays a part of the $\mathcal{PDS}$ that results from the application of ReduceToSpecial to the problem $\lim\limits_{x\to 1}(x+1)*(2*x+3) = 10$. First, ReduceToSpecial decomposes the functions with $+, *$ by applications of the theorems $LIM+$ and $LIM*$. Then, applications of $LIMC$ and $LIMV$ solve the remaining limit goals. All goals with equations on meta-variables are closed by TELLCS-B and passed to $\mathcal{C}o\mathcal{SIE}$. When all line-tasks are closed, then $\mathcal{C}o\mathcal{SIE}$ provides the suitable bindings for the meta-variables (i.e., $mv_4 :=^b 1$, $mv_3 :=^b 1$, $mv_1 :=^b 2$, $mv_2 :=^b 5$).

Another interesting limit theorem in ΩMEGA's database is the Squeeze-Theorem (see [12]). The theorem states that if a function g is squeezed at point c between the two functions f and h and if f and h have the limit l at c, then g has the limit l at c.

$L_8.$	$\vdash mv_3 \doteq 1$	(TellCS-B)
$L_7.$	$\vdash mv_4 \doteq 1$	(TellCS-B)
$L_6.$	$\vdash mv_4 + mv_3 \doteq mv_1$	(TellCS-B)
$L_5.$	$\vdash \lim_{x \to 1} 1 = mv_4$	(ApplyAss-B L_7 $(LIMC)$)
$L_4.$	$\vdash \lim_{x \to 1} x = mv_3$	(ApplyAss-B L_8 $(LIMV)$)
$L_3.$	$\vdash mv_1 * mv_2 \doteq 10$	(TellCS-B)
$L_2.$	$\vdash \lim_{x \to 1} 2 * x + 1 = mv_2$	$(Open)$
$L_1.$	$\vdash \lim_{x \to 1} x + 1 = mv_1$	(ApplyAss-B L_4 L_5 L_6 $(LIM+)$)
$Thm.$	$\vdash \lim_{x \to 1} (x+1) * (2 * x + 3) = 10$	(ApplyAss-B L_1 L_2 L_3 $(LIM*)$)

Figure 8.13: ReduceToSpecial proof for $\lim_{x \to 1} (x+1) * (2 * x + 3) = 10$

$$\begin{array}{ll}\text{Squeeze-Theorem:} & \forall c. \forall l. \forall g. \\ & \quad (\exists f. \exists h. (\forall x_1. (x_1 < c) \Rightarrow (f(x_1) < g(x_1))) \\ & \qquad \wedge (\forall x_2. (x_2 > c) \Rightarrow (g(x_2) < h(x_2))) \\ & \qquad \wedge \lim_{x \to c} f(x) = l \wedge \lim_{x \to c} h(x) = l) \\ & \quad \Rightarrow \lim_{x \to c} g(x) = l \end{array}$$

We exemplify the application of this theorem with the problem $\lim_{x \to 0} sin(x) = 0$. Figure 8.13 depicts a part of the created $\mathcal{PDS}$. When invoked on the problem, then ReduceToSpecial applies the Squeeze-Theorem. This results in the complex goal in L_1, which is the premise of the Squeeze-Theorem instantiated with the elements of the problem at hand. The decomposition of this goal by NormalizeLineTask results in the goals $\lim_{x \to 0} mv_h(x) = 0$ in L_2, $\lim_{x \to 0} mv_f(x) = 0$ in L_3, $sin(c_{x_2}) < mv_h(c_{x_2})$ in L_4, and $mv_f(c_{x_1}) < sin(c_{x_1})$ in L_5, where mv_f is a meta-variable for the function f and mv_h is a meta-variable for the function h, as well as in the hypotheses $c_{x_1} < 0$ in L_7 and $c_{x_2} > 0$ in L_6.

$L_7.$	L_7	$\vdash c_{x_1} < 0$	(Hyp)
$L_6.$	L_6	$\vdash c_{x_2} > 0$	(Hyp)
$L_5.$	L_7	$\vdash mv_f(c_{x_1}) < sin(c_{x_1})$	(ApplyAss-B$_{\{mv_f :=^b -\lvert x \rvert\}}$)
$L_4.$	L_6	$\vdash sin(c_{x_2}) < mv_h(c_{x_2})$	(ApplyAss-B$_{\{mv_h :=^b \lvert x \rvert\}}$)
$L_3.$		$\vdash \lim_{x \to 0} mv_f(x) = 0$	$(Open)$
$L_2.$		$\vdash \lim_{x \to 0} mv_h(x) = 0$	$(Open)$
$L_1.$		$\vdash \exists f. \exists h.$ $(\forall x_1. (x_1 < 0) \Rightarrow (f(x_1) < sin(x_1)))$ $\wedge (\forall x_2. (x_2 > 0) \Rightarrow (sin(x_2) < h(x_2)))$ $\wedge \lim_{x \to 0} f(x) = 0 \wedge \lim_{x \to 0} h(x) = 0$	(NormalizeLineTask L_2 L_3 L_4 L_5)
$Thm.$		$\vdash \lim_{x \to 0} sin(x) = 0$	(ApplyAss-B L_1 $(Squeeze)$)

Figure 8.14: ReduceToSpecial proof for $\lim_{x \to 0} sin(x) = 0$

Crucial for the following proof planning process is the detection of suitable instantiations for mv_f and mv_h that satisfy the "constraints" in L_2, L_3, L_4, L_5. ReduceToSpecial introduces instantiations for mv_f and mv_h by applying further theorems. It closes L_5 and L_4 by applications of the theorems $\forall x. sin(x) \leq |x|$ and $\forall x. -|x| \leq sin(x)$ from ΩMEGA's database. These steps introduce the bindings $mv_f :=^b -|x|$ and $mv_h :=^b |x|$ into the strategic proof plan.[9] We indicate the introduction of these bindings in the justifications of the lines L_4 and L_5 in Figure 8.14.

[9] These bindings are created during the application of ApplyAss-B when the theorems $\forall x. sin(x) \leq |x|$ and $\forall x. -|x| \leq sin(x)$ are matched with the goals in L_5 and L_4. They are part of the resulting method-actions of ApplyAss-B.

With respect to these bindings the formulas of L_2 and L_3 become $\lim_{x \to 0} |x| = 0$ and $\lim_{x \to 0} -|x| = 0$, respectively. ReduceToSpecial fails to solve these problems, but SolveInequality can solve them by constructing ϵ-δ-subproofs.

The Squeeze-Theorem opens a Pandora's box since it is applicable again to its own premises (i.e., in the example in Figure 8.14 ReduceToSpecial could apply the Squeeze-Theorem again to the subgoals in L_2 and L_3 etc.). Thus, the application of the Squeeze-Theorem has to be controlled. A control rule in ReduceToSpecial prefers the two inequality goals resulting from the application of the Squeeze-Theorem before the two limit subgoals. This control rule guarantees that the limit subgoals are tackled only if the two inequality subgoals are closed by theorem applications that instantiate the function meta-variables for f and h.

The extraction of relevant knowledge from the database is a general problem in automated theorem proving. When ReduceToSpecial would check all theorems in ΩMEGA's database, then the check for applicable theorems would overload the system. Hence, a control rule restricts the set of candidate theorems in ReduceToSpecial. Currently, this control rule suggests only the theorems stated in the theory of the current problem. Because of this very inflexible restriction, which encodes no mathematical knowledge or praxis, we had to add the theorems $\forall x_{\bullet} sin(x) \leq |x|$ and $\forall x_{\bullet} - |x| \leq sin(x)$ temporarily to the limit theory in order to test ReduceToSpecial on problems such as $\lim_{x \to 0} sin(x) = 0$. That is, the successful application of ReduceToSpecial currently depends on the location of suitable theorems in the limit theory. We are examining the ΩANTS mechanism as a mediator between ΩMEGA's knowledge base and proof planning (first results are reported in [20]) to overcome the theorem retrieval problem. The mediator supports the idea of semantically guided retrieval of mathematical knowledge (theorems, definitions) from the database.

The combination of ReduceToSpecial and SolveInequality can solve several problems from [12] that cannot be solved by SolveInequality alone, for instance, example 4.2.8(c) $\lim_{x \to 0} cos(x) = 1$ and example 4.2.8(f) $\lim_{x \to 0} x * sin(\frac{1}{x}) = 0$ (when theorems of sin and cos are added into the limit theory).

8.4 Results and Discussion

This chapter presented the application of MULTI to the limit domain. MULTI can solve all problems that PLAN can solve[10] and it successfully plans various problems that are beyond the capabilities of PLAN. In particular, MULTI can solve problems that require eager meta-variable instantiations as well as problems that require meta-reasoning on failures to introduce case-splits, to speculate lemmas, and to guide goal-directed backtracking.

The discussed speculation of lemmas is not possible in PLAN since it does not create and maintain suitable information on failures such as the failure records of MULTI. All other problems are beyond the capabilities of PLAN since it cannot flexibly combine planning, backtracking, and meta-variable instantiation based on meta-reasoning.

We conclude the chapter with a discussion of related work and an evaluation of the realized proof planning approach.

[10] In particular, all challenge problems that BLEDSOE proposed in 1990 [28], among them the limit theorems LIM+, LIM-, LIM*, the theorems Continuous+, Continuous-, Continuous*, $\lim_{x \to a} x = a$, and $\lim_{x \to a} c = c$ (see [172]).

8.4.1 Related Work

Related Work on Proving Limit Theorems

Some of the knowledge encoded in the methods of the SolveInequality strategy is similar to ideas implemented in the theorem prover IMPLY [29] (see also section 2.1.3) developed by BLEDSOE. For instance, COMPLEXESTIMATE-B is inspired by BLEDSOE's limit heuristic. BLEDSOE and HINES developed a resolution-based prover for inequalities [31], which can prove, for instance, the Continuous+ problem. BEESON worked on ϵ-δ-proofs automatically created by the systems MATHPERT and WEIERSTRASS [14]. All these systems rely on special-purpose routines that are implemented into the systems. As opposed thereto, only the strategies, methods, and control rules are domain-specific in ΩMEGA's knowledge-based proof planning, the representational techniques and reasoning procedures are general-purpose.

In [172], MELIS and SIEKMANN describe how to tackle limit theorems with PLAN and compare it with the application of the automated theorem prover OTTER to some limit problems. With a particular control setting OTTER can solve a simple version of LIM+. However, this setting is tailored to LIM+ and does not work for LIM* or other limit theorems. In auto-mode OTTER is not able to prove the simple version of LIM+. In contrast, our strategies, methods, and control rules cover the mathematical knowledge in a form that is general enough to solve all limit problems in Appendix C and many similar theorems that could be formulated.

The LIM+ problem was also proved in CLAM [230] with a special heuristic called *colored rippling*. But LIM* and other theorems of the limit domain turned out to be too difficult for CLAM.

Related Work on Failure Reasoning

Failure reasoning in the proof planner CLAM is closely related to the lemma speculation and the introduction of case-splits in MULTI. Since a detailed comparison of the failure reasonings requires some technical details of CLAM we shall discuss it in the subsequent section 8.4.2.

The speculation of residue lemmas has something in common with HUETS constrained resolution [120]. Since unification is undecidable in higher-order logics constrained resolution intertwines resolution steps with unification. Instead of solving the unification problem $t = t'$ as a precondition of a resolution step, the resolution step is performed and $t = t'$ becomes part of the resolution problem. This process is difficult to control since the introduced unification residue $t = t'$ can be as difficult to solve as the rest of the proof. We also intertwine unification with the main proof process by speculating unification residues as lemmas. But, as opposed to constrained unification, we strictly control the speculation of the lemmas since we allow only for such lemmas that are directly accepted by $\mathcal{C}o\mathcal{SIE}$.

Related to goal-directed backtracking in MULTI is the goal-directed reasoning in elaborate blackboard systems such as HEARSAY-III and BB1 (e.g., see [64, 126] and discussion in section 6.3.1). One approach to integrate goal-directed reasoning in blackboard systems is the construction (and modification) of meta-plans of highly desirable knowledge source applications that guide the following solution process [75]. When a highly desirable knowledge source is not applicable, then reasoning on the failure can suggest the invocation of knowledge sources that unblock the desired knowledge source. When performing goal-directed backtracking, we do not construct meta-plans of strategy applications but we also exploit knowledge of when the application of particular strategies is highly desirable and how to unblock a highly desirable but blocked strategy.

8.4.2 Failure Reasoning in CLAM

In the following, we shall first describe the use of critics in CLAM and then compare failure reasoning with critics with our failure reasoning encoded in control rules.

Critics in CLAM

Bundy and Ireland propose critics as a means to patch failed proof attempts by exploiting information on failures in [122] and [123]. The motivation for the introduction of critics is similar to our motivation for failure reasoning: failures in the proof planning process, in particular, failures occurring after partially successful operations, often hold the key to discover a solution proof plan.

Critics in CLAM extend the hierarchy of inference rules, tactics, and methods. They are introduced in order to complement proof methods. A critic is associated with one method and captures patchable exceptions to the application of the method. Since the application of a method can fail in various ways, each method may be associated with a number of critics. Critics are expressed in terms of preconditions and patches. The preconditions analyse the reasons why the method has failed to apply. The proposed patch suggests a change to the proof plan. This change can be a manipulation of the whole proof plan or the change can be a local manipulation of goals.

To describe the failure reasoning in CLAM we have to consider the construction of inductive proofs in CLAM in some detail. Proof construction in CLAM relies on the domain-independent rippling heuristic [43, 121]. The rippling heuristic is based upon the observation that the induction hypothesis is syntactically similar to the induction conclusion. In order to derive the induction conclusion from the induction hypothesis the `ripple` method tries to rewrite the induction conclusion, such that the induction hypothesis can be used. The `ripple` method iterates over the `wave` method, which applies conditional rewrite rules of the form $Conds \rightarrow (LHS \Rightarrow RHS)$, where LHS is the left hand side, RHS is the right hand side, and $Conds$ are the conditions of the rewrite rule. When $Hyps$ and $Conc$ denote the current hypotheses and the conclusion, respectively, then the preconditions of the `wave` method are:[11]

1. There is a subterm Sub of the conclusion $Conc$, which should be rewritten.
2. There is a conditional rewrite rule $Conds \rightarrow (LHS \Rightarrow RHS)$ such that LHS matches with Sub.
3. The conditions $Conds$ are satisfied by the hypotheses $Hyps$ (i.e., $Hyps \vdash Conds$ is a tautology).

The application of the `wave` method fails, when one of its preconditions is not satisfied. Bundy and Ireland realized two patches for the method, which are implemented as critics associated with the method:

1. A failure of precondition 2, i.e., there is no rewrite rule that can be applied, triggers the `lemma-discovery` critic. The preconditions for the application of this critic are: (1) precondition 1 of the `wave` method holds and (2) preconditions 2 and 3 fail. The patch of the critic involves the speculation and proof of a rewrite rule to unblock this situation. This process may involve backtracking, when a speculated rewrite rule cannot be proved.

[11] Actually, there are different `wave` methods for different kinds of rippling (e.g., longitudinal-rippling and transverse-rippling), which have some more preconditions that differ slightly among the different `wave` methods, see [43, 123] for details. For the sake of simplicity we discuss here only the relevant preconditions.

2. A failure of precondition 3, i.e., the condition of a matching rewrite rule is not satisfied in the current context, triggers the `missing-condition` critic. The preconditions for the application of this critic are: (1) precondition 1 of the `wave` method holds, (2) precondition 2 of the `wave` method holds with respect to a rewrite rule $Conds \rightarrow (LHS \Rightarrow RHS)$, and (3) precondition 3 fails for $Conds$. The patch of the critic is to perform a case analysis based upon the unprovable conditions $Conds$.

These two critics are tailored to the possible failures of the application of the `wave` method. The general ideas behind the critics are:

Lemma Speculation: When no methods are applicable with respect to the current context, the controlled speculation (and the proof) of new lemmas can unblock the proof planning process.

Case Analysis: Splitting a problem into different cases can unblock the proof planning process, when no methods are applicable.

Bundy and Ireland describe also critics of other methods that patch the selection of the induction schemata and generalize conjectures in order for an inductive proof to succeed (see [123]).

Comparison with Failure Reasoning in Multi

The situations that trigger lemma speculation and case-splits in CLAM and Multi are very similar: missing premises in the current context (i.e., missing rewrite rules in CLAM or missing supports in Multi) trigger lemma speculation; unprovable premises of conditional facts from the context (i.e., conditional rewrite rules in CLAM or conditional supports in Multi) cause case-splits. However, the critics mechanism in CLAM and failure reasoning in Multi considerably differ not only in minor technical issues but also in their conceptual design.

Critics in CLAM are an extra concept introduced for failure reasoning. A critic reasons on failures of the one method it is directly associated with, i.e., it reasons on failing preconditions of the method. Part of a critic is a patch of the failure. Technically, this patch is a special procedure that can change the complete proof plan.

In contrast, failure reasoning in Multi is conducted by control rules. The control rules are not associated with a particular method but rather test for particular situations that can occur during the proof planning process (independent from which strategy or method caused the situation). The control rules reason on the current proof plan and on all other available information such as the history. The patch of a failure is not implemented into special procedures but is carried out by methods and strategies whose application is suggested by the control rules.

The advantage of the Multi approach is that control rules allow for method- and strategy-independent reasoning on failures. For instance, the control rule `choose-equation-residues`, which guides the lemma speculation can deal with failing *unify* and *matching* application conditions of any employed method. It is domain-independent since it could be employed in cooperation with other constraint solvers similar to the cooperation with $\mathcal{C}o\mathcal{SIE}$ described in section 8.2.2.

We decided to realize patches in Multi by control rules that guide the application of existing strategies and methods since procedural patches are difficult to maintain. Both the introduction and the deletion of a patch for a desired manipulation requires the implementation of special procedures. For complex proof plan manipulations the cooperation of several methods and strategies can be necessary and has to be guided by several control rules. For instance, when performing case analysis, Multi has to backtrack the application of the conditional support. Afterwards, it has to introduce the case-split and finally it has to replay the backtracked parts again (in order to avoid to prove again from the scratch). The necessary failure reasoning and the knowledge of how to patch this failure is distributed among three control rules: one strategic control rule that guides the backtracking, one control rule that guides the case split, and one control rule that guides the replay of the backtracked parts. Although the failure reasoning is distributed we see the three involved control rules as one meta-reasoning entity that is distributed for technical reasons.

8.4.3 Evaluation of the Proof Planning Approach

Knowledge-based proof planning relies on the acquisition, formalization, and use of domain-specific knowledge in methods, control rules, and strategies. However, there is the constant danger to acquire over-specific knowledge as Bundy points out:

> *A new method or critic may originally be inspired by only a handful of examples. There is a constant danger of producing methods and critics that are too find tuned to these initial examples. This can arise both from a lack of imagination in generalizing from the specific situation and from the temptation to get quick results in automation. Such over-specificity leads to a proliferation of methods and critics with limited applicability.*
>
> Bundy, [42]

Bundy suggests in [42] and [39] the criteria *generality* and *parsimony* to evaluate the appropriateness of proof planning methods and critics. Generality means that each method or critic should apply successfully in a wide range of situations, whereas parsimony means that a few methods should generate a large number of proofs.

These criteria of Bundy do not consider mathematical content, which is an important issue in knowledge-based proof planning. The methods, control rules, and strategies in knowledge-based proof planning should be rich in mathematical content. Thus, the art of knowledge-based proof planning is to acquire domain knowledge that, on the one hand, comprises meaningful mathematical techniques and powerful heuristic guidance, and, on the other hand, is general enough to tackle a broad class of problems.

In the following, we shall evaluate proof planning limit theorems with Multi. We discuss the amount of mathematical and domain-specific knowledge in strategies, methods, and control rules and discuss how general they are. We discuss generality not only in the sense of Bundy, that is, to how many problem classes a concrete strategy, method, or control rule applies. Rather, we discuss also how general the encoded principle is and how it can be transfered to other domains.

SolveInequality

The approach to tackle inequality problems with the SolveInequality strategy fits into a much more general heuristic strategy described by Schoenfeld :

> *In a problem 'to find' or 'to construct', it may be useful to assume that you have the solution to the given problem. With the solution (hypothetically) in hand, determine the properties it must have. Once you know what those properties are, you can find the object you seek.*
>
> Schoenfeld, [209] p. 23

When tackling inequality problems, SolveInequality assumes that solutions for existentially quantified variables exist (e.g., for the δ in ϵ-δ-proofs) and substitutes the existentially quantified variables by meta-variables. Afterwards, it collects constraints on the introduced meta-variables in $\mathcal{C}o\mathcal{SIE}$, which at the end computes instantiations for the meta-variables.

Now that we know that SolveInequality fits into the general strategy "assume, collect properties, then compute", could we encode a general version of this strategy that can tackle various domains and subsumes SolveInequality? Probably not, since, as SCHOENFELD points out, such a general heuristic strategy alone provides no adequate information on how to use this strategy in a concrete case.

> [...] *that a typical heuristic strategy is very broadly defined — too broadly, in fact, for the description of the strategy to serve as a useful guide to its implementation.*
>
> Schoenfeld,[209] pp. 70 and 72

Rather, such general strategies have to be filled with domain-specific knowledge such that the general strategy is only a summary label for a class of substrategies for different domains:

> [...] *the successful implementation of heuristic strategies in any particular domain often depends heavily on the possession of specific subject matter knowledge.*
> [...] *More often than not, a capsule description of a strategy is a summary label that includes under it a class of more precise substrategies that may be only superficially related.*
>
> Schoenfeld,[209] pp. 92 and 95

Thus, in the sense of SCHOENFELD, SolveInequality is a substrategy of the general strategy "assume, collect properties, then compute". It instantiates this general principle with the specific knowledge on how to apply it to inequalities over the reals.

The main control rule of SolveInequality, `prove-inequality`, encodes the essential idea of how SolveInequality implements the general principle for inequalities over the reals: reduce complex inequalities to simple inequalities and pass simple inequalities to the connected constraint solver. To tackle complex inequalities `prove-inequality` suggests domain-specific methods such as SIMPLIFY-B, SOLVE*-B, COMPLEXESTIMATE-B, and FACTORIALESTIMATE-B. These methods encode mathematical knowledge of inequalities, real numbers, and the operations $+, -, *, /$ on real numbers. This knowledge is partially contained in the computer algebra system MAPLE that is employed within COMPLEXESTIMATE-B and SIMPLIFY-B. Moreover, `prove-inequality` suggests the methods TELLCS-B, TELLCS-F, and ASKCS-B that interface the constraint solver $\mathcal{C}o\mathcal{SIE}$. These methods do not contain domain-specific mathematical knowledge but provide a domain-independent interface to constraint solvers.

The domain-specific methods of SolveInequality are hardly reusable in another substrategy of "assume, collect properties, then compute" for other domains. However, they could be useful for other problem classes dealing with inequalities over the reals. Currently, the methods TELLCS-B, TELLCS-F, and ASKCS-B interface only $\mathcal{C}o\mathcal{SIE}$. However, they provide general functionalities, namely adding constraints and asking whether a constraint is entailed, that are independent of a concrete constraint solver. Thus, they can be used also in other domains with other constraint solvers (e.g., problems on sets with a constraint solver on sets).

The essence of the control rule `prove-inequality` could be reused in other substrategies of the "assume, collect properties, then compute" strategy for other domains with constraint solvers. In such a domain, the adaption of `prove-inequality` would suggest domain-specific methods to tackle complex expressions of this domain until TELLCS-B, TELLCS-F, and ASKCS-B involve a constraint solver of the domain to handle the simple expressions.

SolveInequality also contains some logic-level methods, for instance, CONTRA-B to perform indirect proofs and DEFNUNFOLD-B and DEFNUNFOLD-F for unfolding of defined concepts. These methods are domain-independent and contain no particular mathematical knowledge. The decision when to perform an indirect proof and which definitions to unfold and which not are difficult problems in theorem proving in general (e.g., see [30, 249, 102] for discussions on unfolding of defined concepts). Their application within SolveInequality is guided by control rules that encode mathematical heuristics. For instance, since the purpose of SolveInequality is to tackle inequalities it only unfolds defined concepts that result in inequalities. This knowledge is encoded in the control rule `select-unfold-defined-concept`, which guides the application of DEFNUNFOLD-B and DEFNUNFOLD-F. The meta-reasoning to guide indirect proofs in the limit domain is discussed in [171].

SolveInequality employs some further control rules that do not encode mathematically meaningful heuristics but deal with technical peculiarities that occur during the search process. As example for such a control rule consider `block-simplify`, which restricts applications of the methods SIMPLIFY-F and SIMPLIFY-B. Both methods employ MAPLE to simplify arithmetic terms. Unfortunately, it turned out that sometimes the application of MAPLE results in more complex terms. To avoid unnecessary complexity and non-terminating cycles of simplification and complication `block-simplify` rejects all applications of SIMPLIFY-F and SIMPLIFY-B that do not simplify the terms.

Altogether, SolveInequality is not restricted to limit problems. Rather, its approach is general enough to tackle also other inequality problems over the reals. However, since we did focus on limit problems so far, the methods of SolveInequality are focused on inequalities with absolute values. To extend the solvability horizon of the strategy some methods are needed that tackle complex inequalities without absolute values, for instance, methods similar to COMPLEXESTIMATE-B or methods that isolate subterms in complex inequalities (isolating x in $(c-x)+a<\epsilon$ results in $x>(c+a)-\epsilon$).[12]

NormalizeLineTask, UnwrapHyp, and ReduceToSpecial

The **PPLANNER** strategies NormalizeLineTask and UnwrapHyp contain only logic-level methods to decompose complex formulas in goals and supports. Thus, they are very general in the sense of BUNDY, but they do not encode any specific mathematical knowledge. However, they implement operations that are important in mathematical problem solving in general since the decomposition of complex goals and the unwrapping of subformulas of complex assumptions is necessary in all mathematical domains where complex statements are composed from primitive ones by logical connectives and quantifiers.

ReduceToSpecial uses only general methods, in particular, a domain-independent method for the application of theorems. However, we had to add some domain-specific control to guide the application of the Squeeze-Theorem. The content of this control is not of mathematical nature, rather it comprises technical knowledge on

[12] An example theorem that requires the handling of complex inequalities without absolute values is the Squeeze-Theorem. Although we employ this theorem when proving problems with the ReduceToSpecial strategy it currently cannot be proved by MULTI.

how to prevent MULTI from the repeated, never-ending application of the Squeeze-Theorem.

INSTMETA Strategies

Similar to the methods TELLCS-B, TELLCS-F, and ASKCS-B the **INSTMETA** strategies InstIfDetermined and ComputeInstFromCS encode no particular mathematical knowledge but provide interface functions to constraint solvers. Although, currently they interface only $\mathcal{C}o\mathcal{SIE}$, they provide functionalities, namely retrieving particular entailed constraints and computation of instantiations, that are independent of a concrete constraint solver. Thus, they could be employed also in other domains.

Failure Reasoning

The described mathematical knowledge to speculate lemmas and to introduce case-splits are general meta-reasoning patterns, promising also for other domains. As evidence for this statement consider that the corresponding critics in CLAM exploit very similar failures in a completely different domain to guide similar proof modifications.

The domain-specific part of the lemma speculation described in section 8.2.2 is the decision of which lemmas are promising and which not. To avoid the speculation of arbitrary lemmas that cannot be proved in the current context, SolveInequality asks $\mathcal{C}o\mathcal{SIE}$ whether it accepts a potential lemma. This exploits the domain-specific information encoded in $\mathcal{C}o\mathcal{SIE}$ as well as the context information passed to $\mathcal{C}o\mathcal{SIE}$ so far. The same approach could be performed in other domains with constraint solvers that contain particular domain knowledge. Other domains maybe provide different kinds of guidance to decide whether lemmas are promising.

The domain-specific part of the case-split introduction discussed in section 8.2.1 is the decision of which cases to consider. In the limit domain, the general case-split $C \vee \neg C$ was sufficient so far to deal with a failing condition C. The case-split $C \vee \neg C$ is domain-independent since it relies only on the *tertium-non-datur* axiom of ΩMEGA's underlying logic. However, it can be necessary to construct domain-specific case-splits. For instance, when C equals $a < b$, then the case-split $a < b \vee a = b \vee a > b$ could be considered. Different domains maybe provide different kinds of domain-specific case-splits.

The goal-directed backtracking discussed in section 8.2.3 is just one particular example of goal-directed reasoning on failures. More generally stated the principle works as follows: Suppose there is a meta-plan (either explicitly constructed somewhere or implicitly encoded in control rules) of the desired solution process, and suppose that a step S of this meta-plan fails. Then, the failure can be analyzed and further steps can be considered in order to unblock S. The concrete pattern (unblock ComputeInstFromCS if there are no further goals) is restricted to the limit domain (and maybe some other domains with constraint solvers). The general principle, however, is a domain-independent, promising meta-reasoning pattern for any domain for which a kind of meta-plan of the desired solution process exists.

Summary

Typical questions of referees of our papers on proof planning are, for instance:

- How many new methods are typically needed when a new chapter in a book is considered?
- How many of the methods can typically be reused, when a new chapter in a

book is considered?

A general answer to those questions is not possible. When extending the domain of proof planning, the crucial question is whether the knowledge acquired so far is sufficient to tackle the new problems.

To illustrate this subtle point consider the following experiences in the limit domain. We started to develop proof planning in the limit domain with examples from chapter 4 and chapter 5 in [12] on the limit of functions and the continuity of functions. On the one hand, we found that the acquired knowledge was not sufficient to deal with several problems in chapter 4 and chapter 5. These problems need additional knowledge about particular functions involved. For instance, Multi can solve some problems on trigonometric functions only with specific knowledge on the functions *sin* and *cos* in some theorems (see section 8.3). Currently, it cannot solve, for instance, problems involving the square-root function since the methods and theorems do not contain appropriate knowledge of this function. On the other hand, we found that with the knowledge acquired for chapter 4 and chapter 5 Multi can solve problems on the derivative of functions without any extensions in form of further methods, control rules, or theorems although this is a new chapter (chapter 6) in [12].

These experiences demonstrate the success and the limitation of the current proof planning for limit problems realized in Multi:

1. The implemented methods, control rules, and strategies are not too fine tuned to our initial examples. In particular, the control rules contain the necessary control knowledge in a form that is general enough to deal also with new problems for which the domain knowledge in the methods and strategies is sufficient.

2. The implemented methods, control rules, and strategies are not sufficient to deal with any limit problems. They are mainly restricted to terms composed of $+, -, *, /, ||$. To deal with further expressions such as square-root requires further specific knowledge.

Chapter 9

The Residue Class Domain

This chapter presents a case study on proof planning for the residue class domain (see section 5.2 for a formal introduction of the residue class domain). The residue class domain consists of the problems given in Table 9.1 for arbitrary residue class structures. We call the problems 1—7 *problems on simple properties* of residue class structures, whereas the problems 8 are called *isomorphism and non-isomorphism problems*.

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) $\exists e{:}_{RS_n}\centerdot Unit(RS_n, \circ, e)$	(b) $\neg\exists e{:}_{RS_n}\centerdot Unit(RS_n, \circ, e)$
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) $\neg Distrib(RS_n, \circ, \star)$
8.	(a) $Iso(RS^1_n, \circ^1, RS^2_m, \circ^2)$	(b) $\neg Iso(RS^1_n, \circ^1, RS^2_m, \circ^2)$

Table 9.1: Problems from the residue class domain.

The chapter is structured as follows. We start in section 9.1 with a description of how Multi creates proof plans for simple property problems. Afterwards, we explain in section 9.2 how the strategies for simple property problems are extended to deal with isomorphism and non-isomorphism problems and introduce further techniques specialized on non-isomorphism problems. Both sections, 9.1 and 9.2, comprise the description of automated exploration modules implemented in Ωmega. The exploration module for simple property problems classifies a given residue class structure in terms of the algebraic entity it forms (i.e., is it a magma, a semi-group, a monoid ...); the exploration module for isomorphism and non-isomorphism problems classifies a set of structures into classes of isomorphic structures. We conclude the chapter with a report on conducted experiments and a discussion of related work. Moreover, we shall evaluate the realized proof planning approach in the residue class domain and compare it with the application of an automated theorem prover to this domain. An overview of the proved theorems in the residue class domain is given in the technical report [164].

9.1 Proof Plans of Simple Property Problems

In order to proof plan simple property problems of a residue class structure we implemented three different **PPlanner** strategies. Each strategy implements a different mathematical proof technique, namely:

1. exhaustive case analysis, realized in the strategy TryAndError,
2. equational reasoning, realized in the strategy EquSolve, and
3. application of theorems, realized in the strategy ReduceToSpecial.

Not all strategies are applicable to all possibly occurring problems. The idea to control the application of these strategies is to employ fast but not always successful strategies first, and if they fail to use slower but more reliable strategies. Since the strategy ReduceToSpecial is generally the fastest to solve a problem and strategy TryAndError is the most reliable of the three strategies, the strategic control rule `fast-before-reliable` orders job offers of these strategies in the order 3 to 1.

Note that the three strategies either succeed to prove a simple property for a residue class structure or fail. Multi does not intertwine these three **PPlanner** strategies in the sense that certain subgoals arising during the application of one strategy can be proved with another technique. Intertwining of **PPlanner** strategies is used when checking whether two structures are isomorphic or not, see section 9.2. However, Multi has to intertwine these **PPlanner** strategies with strategies of **BackTrack** and **InstMeta**, which we shall introduce as we go along.

In the sequel, we first 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. Afterwards, we point out how structures with direct products of residue class sets are formalized and how they are handled by the strategies. We conclude with a discussion of the exploration module, which classifies a given residue class structure in terms of its algebraic category.

9.1.1 Exhaustive Case Analysis

The motivation for the first strategy, called TryAndError, is to implement an exhaustive case analysis, which ideally should be able to solve all types of problems.[1] This technique is possible in our domain since in residue class problems the quantified variables range always over finite domains.

When applied to a simple property problem, TryAndError first expands occurrences of the defined concepts *closed*, *assoc*, *unit*, *inverse*, *divisors*, *commu*, and *distrib* with the method DefnUnfold-B. 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 it has to prove or to refute. The organization of the exhaustive case analysis is guided by the control rule `tryanderror-standard-select` (see Figure 4.4 in section 4.1.3).

TryAndError can proceed in two different ways, depending on whether (1) a universally or (2) an existentially quantified formula has to be proved. Both cases are illustrated in the example proof of the theorem that $\mathbb{Z}_2$ has inverses with respect to the operation $\lambda xy.x\bar{+}y$ and the unit element $\bar{0}_2$, displayed in Figure 9.1.

[1] In our 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 9.3).

L_1.	L_1	$\vdash cl_2(c) \in \mathbb{Z}_2$	(Hyp)
L_2.	L_1	$\vdash c \in \{0,1\}$	(ConResclSet-F L_1)
L_3.	L_3	$\vdash c \doteq 0$	(Hyp)
		$\vdots$	
L_{12}.	L_1, L_3	$\vdash \exists y{:}\mathbb{Z}_2 \centerdot (cl_2(c) \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} cl_2(c) \doteq \bar{0}_2)$	($\exists$IResclass-B L_{11} L_{10})
L_{13}.	L_{13}	$\vdash c \doteq 1$	(Hyp)
L_{14}.	L_1, L_{13}	$\vdash mv \doteq 1$	($\doteq$Reflex-B$_{\{mv:=^b 1\}}$)
L_{15}.	L_1, L_{13}	$\vdash mv \in \{0,1\}$	($\vee$IR-B L_{14})
L_{16}.	L_1, L_{13}	$\vdash 0 \doteq 0$	($\doteq$Reflex-B)
L_{17}.	L_1, L_{13}	$\vdash 0 \doteq 0$	($\doteq$Reflex-B)
L_{18}.	L_1, L_{13}	$\vdash (1+c)\ \mathit{mod}\ 2 \doteq 0\ \mathit{mod}\ 2$	(SimplifyNum-B L_{13} L_{16})
L_{19}.	L_1, L_{13}	$\vdash (c+1)\ \mathit{mod}\ 2 \doteq 0\ \mathit{mod}\ 2$	(SimplifyNum-B L_{13} L_{17})
L_{20}.	L_1, L_{13}	$\vdash (c+1)\ \mathit{mod}\ 2 \doteq 0\ \mathit{mod}\ 2\ \wedge$	($\wedge$I-B L_{18} L_{19})
		$(1+c)\ \mathit{mod}\ 2 \doteq 0\ \mathit{mod}\ 2$	
L_{21}.	L_1, L_{13}	$\vdash (cl_2(c) \bar{+} cl_2(mv) \doteq \bar{0}_2) \wedge cl_2(mv) \bar{+} cl_2(c) \doteq \bar{0}_2)$	(ConCongCl-B L_{20})
L_{22}.	L_1, L_{13}	$\vdash \exists y{:}\mathbb{Z}_2 \centerdot (cl_2(c) \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} cl_2(c) \doteq \bar{0}_2)$	($\exists$IResclass-B L_{21} L_{15})
L_{23}.	L_1	$\vdash \exists y{:}\mathbb{Z}_2 \centerdot (cl_2(c) \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} cl_2(c) \doteq \bar{0}_2)$	($\vee$E**-B L_2 L_{12} L_{22})
L_{24}.		$\vdash \forall x{:}\mathbb{Z}_2 \centerdot \exists y{:}\mathbb{Z}_2 \centerdot (x \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} x \doteq \bar{0}_2)$	($\forall$IRescalss-B L_{23})
L_{25}.		$\vdash inverse(\mathbb{Z}_2, \lambda xy \centerdot x \bar{+} y, \bar{0}_2)$	(DefnUnfold-B L_{24})

Figure 9.1: Proof constructed by the TryAndError strategy.

In case (1), TryAndError performs a split over all the elements in the set $\mathbb{Z}_2$ and proves the property for every single element separately. We exemplify this in the proof of the universally quantified formula in line L_{24}. An application of the method $\forall$IRescalss-B to L_{24} yields the lines L_{23}, L_1, and L_2. $\forall$IRescalss-B is a method to decompose universally quantified goals whose variables range over residue class sets. It is dual to $\exists$IResclass-B that has been explained in section 4.1.1. The disjunction contained in L_2 ($c \in \{0,1\}$ can be viewed as $c \doteq 0 \vee c \doteq 1$) triggers the first case-split with the application of the method $\vee$E**-B (explained in section 4.1.3). Subsequently, Multi tries to prove the goal in line L_{23} twice: once in line L_{12} assuming $c \doteq 0$ (in line L_3) and once in L_{22} assuming $c \doteq 1$ (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 proof this is, for instance, done after the application of the method $\exists$IResclass-B to L_{22}, which yields the lines L_{15} and L_{21} and introduces the meta-variable mv. The case analysis is performed by successively choosing different possible values for mv with the $\vee$IR-B and $\vee$IL-B methods that split disjunctive goals into the left or right disjunct, respectively, and the $\doteq$Reflex-B method, which closes goals of the form $t_1 \doteq t_2$. Applications of $\doteq$Reflex-B introduce then the unifier of t_1 and t_2 as new bindings. In our example the application of $\vee$IR-B reduces $mv \in \{0,1\}$ in L_{15} to $mv \doteq 1$ in L_{14} ($mv \in \{0,1\}$ can be viewed as $mv \doteq 0 \vee mv \doteq 1$) and the application of $\doteq$Reflex-B to L_{14} introduces the binding $mv :=^b 1$ into the strategic proof plan. We indicate the introduction of the binding by attaching it to the justification of line L_{14}. For a selected binding TryAndError can then either finish the proof (i.e., can close the remaining open goals with respect to this binding) or — if the proving attempt fails — it has to test the next possible binding.

After eliminating the quantifiers and introducing the case-splits the TryAndError strategy reduces all remaining statements on residue and congruence classes to statements on integers using the ConCongCl-B method. These are solved by numerical simplification and basic equational reasoning through the methods SimplifyNum-B and $\doteq$Reflex-B.

Note that in our example we describe the proof planning process in progress. Hence, we introduce meta-variables, when they arise. When there is a binding for a meta-variable, we use in the proof lines created after the introduction of the binding the instantiation of the meta-variable in order to clarify the following computations.

Thus, in the proof plan in Figure 9.1 the lines L_{15}, L_{14}, and L_{21} contain occurrences of mv. From L_{20} on we use occurrences of the instantiation 1 for mv instead.

9.1.1.1 Meta-Reasoning on Backtracking

Meta-variables and their instantiations cause dependencies among goals that share some meta-variables. As a general example consider two goals G and G' that contain both a meta-variable mv. Now assume that MULTI first creates a proof plan for G in which it binds mv in such a way that it afterwards fails to solve G'. Without meta-reasoning on the failure MULTI would employ the standard **BACKTRACK** strategy BackTrackActionToTask and would remove G'. However, when there are different possibilities to instantiate mv in a subplan for G, then the actual problem may is not G' but the selection of the right instantiation for mv. That is, MULTI should delete part of the subplan for G to introduce another subplan that instantiates mv differently, rather than to delete G'.

We formalized the meta-reasoning to deal with those situations in the strategic control rule `prefer-binding-deletion`. This control rule analyzes a failure and, if it finds that the failure was caused by a wrong binding, it prefers job offers of the **BACKTRACK** strategy BackTrackLastBinding before job offers of BackTrackActionToTask. Let T be the task for which a failure occurs and A the action that introduced T. Then, BackTrackActionToTask deletes A, whereas BackTrackLastBinding deletes actions introduced after A that introduced new bindings.

We illustrate the application of BackTrackLastBinding with the example in Figure 9.1. TryAndError has to organize the successive check of each possible binding for the meta-variable mv introduced by the application of the method $\exists$IRESCLASS-B to L_{22}. This yields the open lines L_{15} and L_{21}, which both contain mv. mv is either 0 or 1 as given in line L_{15}. Assume that TryAndError first reduces L_{15} to $mv \doteq 0$ by an application of $\vee$IL-B and then closes $mv \doteq 0$ by $\doteq$REFLEX-B. This introduces the new binding $mv :=^{b} 0$. TryAndError would fail to close afterwards the goal L_{21} with respect to this binding, since mv is supposed to be the inverse of $\bar{1}_2$ in $\mathbb{Z}_2$, which is again $\bar{1}_2$.

When TryAndError fails on L_{21} in our example, then `prefer-binding-deletion` guides the application of BackTrackLastBinding which deletes the subplan for L_{15} including the binding for mv. Afterwards, TryAndError applies $\vee$IR-B instead of $\vee$IL-B, which reduces L_{15} to $mv \doteq 1$ (L_{14} in Figure 9.1). The following application of $\doteq$REFLEX-B yields the binding $mv :=^{b} 1$ with respect to which L_{21} can be closed as given in Figure 9.1.

9.1.1.2 Meta-Variable Instantiation

To minimize the search for a suitable instantiation of a meta-variable, which can become very tedious for large residue class sets or for nested meta-variables, TryAndError cooperates with the **INSTMETA** strategy ComputeInstbyCasAndMG. ComputeInstbyCasAndMG employs the computer algebra systems MAPLE and GAP as well as the model generator SEM to compute instantiations.

When applied to an instantiation-task, ComputeInstbyCasAndMG first analyzes what kind of instantiation is needed. To do so, it checks the proof lines that contain occurrences of the meta-variable of the given instantiation-task for "constraints" that determine the needed kind of instantiation. For instance, for the meta-variable mv in Figure 9.1 ComputeInstbyCasAndMG finds the proof line L_{21} and analyzes that mv has to be instantiated by the inverse of $\bar{1}_2$ in $\mathbb{Z}_2$. After analyzing the needed kind of instantiation, ComputeInstbyCasAndMG employs the computer algebra systems and the model generator to compute the concrete instantiation.[2]

To employ the computer algebra systems ComputeInstbyCasAndMG constructs a multiplication table with respect to the found residue class set and operation. It checks the closure property directly with this multiplication table. If the computed multiplication table is closed under the respective operation, then ComputeInstbyCasAndMG passes it to GAP to construct the appropriate magma in GAP. Afterwards, ComputeInstbyCasAndMG can employ GAP to test for associativity and to compute the unit element and inverses for the single elements. Most test functions return useful results in both the positive and the negative case: That is, for instance, if GAP can compute a unit element for a given magma, this element is returned. In case GAP fails to find a unit element, the multiplication table is used to determine a set of elements that suffice to refute the existence of a unit element for the given magma. A special case is the failure of the test for associativity, since there MAPLE is employed to compute a particular solution for the associativity equation. If such a non-general solution exists, it is exploited to determine a triple of elements for which associativity does not hold.

When employing SEM, ComputeInstbyCasAndMG also constructs a multiplication table with respect to the found residue class set and operation. The actual call to SEM consists of this multiplication table together with the problem at hand. The multiplication table for n elements is encoded as a set of n^2 equations of the form $a \circ b = c$. To obtain, for example, a unit element SEM is asked to compute a model for the equations $x * e = x$ and $e * x = x$, where x is a free variable and e is an unspecified constant function for which a model is computed.

The cooperation between TryAndError and ComputeInstbyCasAndMG is guided by the control rule `interrupt-if-inst-from-cas-or-mg`, which is part of TryAndError. This control rule interrupts TryAndError for occurring meta-variables and poses a demand to first invoke ComputeInstbyCasAndMG on the instantiation-task of the meta-variable.

The cooperation with ComputeInstbyCasAndMG is not necessary for the success of TryAndError. However, if ComputeInstbyCasAndMG can provide suitable instantiations for meta-variables, then the problems are simplified considerably. Even if ComputeInstbyCasAndMG succeeds, the strategy TryAndError has the major disadvantage that it has to exhaustively construct subproofs for all cases resulting from universal quantifications, which can result in lengthy proofs for large residue class sets.

9.1.2 Equational Reasoning

The aim of the second strategy, called EquSolve, is to use equational reasoning as much as possible to prove properties of residue classes. Its application condition states that EquSolve can tackle only problems that can be reduced to equations (i.e., it cannot tackle problems involving the closure property or refutations of a property).

Similarly to the TryAndError strategy, EquSolve converts statements on residue classes into corresponding statements on integers. But instead of checking the validity of the statements for all possible cases, it tries to solve occurring equations

[2] Because of historical reasons (we did first implement the connection to the computer algebra systems), ComputeInstbyCasAndMG first employs the computer algebra systems and afterwards SEM only if the computer algebra systems fail to provide a suitable solution. Currently, we are working on a concurrent implementation that runs SEM and the computer algebra systems in a competitive manner.

$L_1.$	$L_1 \vdash c'_1 \in \mathbb{Z}_2$	(Hyp)
$L_2.$	$L_1 \vdash c \in \{0,1\}$	(ConResclSet-F L_1)
$L_{15}.$	$L_1 \vdash mv \in \{0,1\}$	(Weaken-B L_2)
$L_{18}.$	$L_1 \vdash (mv + c)\ mod\ 2 \doteq 0\ mod\ 2$	(SolveEquation-b$_{\{mv:=^b c\}}$)
$L_{19}.$	$L_1 \vdash (c + mv)\ mod\ 2 \doteq 0\ mod\ 2$	(SolveEquation-b)
$L_{20}.$	$L_1 \vdash (c + mv)\ mod\ 2 \doteq 0\ mod\ 2\ \wedge$	($\wedge$I-B L_{19} L_{18})
	$(mv + c)\ mod\ 2 \doteq 0\ mod\ 2$	
$L_{21}.$	$L_1 \vdash (cl_2(c) \bar{+} cl_2(mv) \doteq \bar{0}_2) \wedge (cl_2(mv) \bar{+} cl_2(c) \doteq \bar{0}_2)$	(ConCongCl-B L_{20})
$L_{22}.$	$L_1 \vdash \exists y{:}\mathbb{Z}_{2\bullet}((cl_2(c) \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} cl_2(c) \doteq \bar{0}_2))$	($\exists$IResclass-B $L_{21} L_{15}$)
$L_{24}.$	$\vdash \forall x{:}\mathbb{Z}_{2\bullet} \exists y{:}\mathbb{Z}_{2\bullet}((x \bar{+} y \doteq \bar{0}_2) \wedge (y \bar{+} x \doteq \bar{0}_2))$	($\forall$IRescalss-B L_{23})
$L_{25}.$	$\vdash inverse(\mathbb{Z}_2, \lambda xy_\bullet x \bar{+} y, \bar{0}_2)$	(DefnUnfold-B L_{24})

Figure 9.2: Proof constructed by the EquSolve strategy.

in a general way. We illustrate EquSolve's approach with a proof of the example theorem from section 9.1.1 $inverse(\mathbb{Z}_2, \lambda xy_\bullet x \bar{+} y, \bar{0}_2)$, displayed in Figure 9.2.

In the beginning (lines L_{25} through L_{20}), the construction of the proof is nearly analogous to the one in the preceding section. The only exception is that no case-splits are carried out after the applications of $\forall$IRescalss-B and $\exists$IResclass-B. Instead EquSolve obtains two equations in the lines L_{18} and L_{19} which it can generally solve using the SolveEquation-b method. This method is applicable, if Maple can compute a solution of the given equation. In case the equation in question contains meta-variables, the solution Maple computes can bind these meta-variables. In our example, the application of SolveEquation-b to L_{18} — the first application of SolveEquation-b — introduces a binding for mv, namely $mv{:=}^b\, c$, which is indicated in the justification of L_{18}. The binding for mv changes the formulas in the remaining open goals L_{19} and L_{15} to $(c+c)\ mod\ 2 \doteq 0\ mod\ 2$ and $c \in \{0,1\}$. EquSolve closes L_{19} by another application of SolveEquation-b. Since L_{15} equals meanwhile L_2 it is closed from this line by an application of Weaken-B.

As opposed to the TryAndError strategy, the proofs EquSolve constructs are independent of the size of the residue class set. But the strategy can be applied only to some of the occurring problems. Whether EquSolve succeeds to solve a given problem depends on whether the equations have solutions and whether Maple can solve them.

9.1.3 Applying Theorems

In order to incorporate the application of already proved theorems we use the strategy ReduceToSpecial known from the limit domain also to tackle residue class problems.

To do so, we had to slightly extend ReduceToSpecial with further methods to apply theorems besides the primary method ApplyAss-B. To ensure termination ApplyAss-B uses first-order matching with α-equality on λ-abstractions. For the application of some of the theorems of the residue class domain we actually need higher-order matching. In order to stay decidable, we decided against using a general method that applies theorems with higher-order matching. Instead, we added some methods that decide the applicability of certain theorems with specialized algorithms, for instance, the method ReduceClosed-B.

We illustrate the application of ReduceToSpecial with the proof for the theorem $closed(\mathbb{Z}_5, \lambda x, y_\bullet (x \bar{*} y) \bar{+} \bar{3}_5)$ given in Figure 9.3. The following are the theorems involved:[3]

[3] Similarly, our database contains theorems suitable for associativity, unit element, inverses, and divisor problems.

$L_3.$	$\vdash \bar{3}_5 \in \mathbb{Z}_5$	$(InResclSet)$
$L_4.$	$\vdash 5 \in \mathbb{Z}$	$(InInt)$
$L_5.$	$\vdash closed(\mathbb{Z}_5, \lambda xy.x)$	$(ApplyAss\ ClosedFV)$
$L_6.$	$\vdash closed(\mathbb{Z}_5, \lambda xy.y)$	$(ApplyAss\ ClosedSV)$
$L_7.$	$\vdash 5 \in \mathbb{Z}$	$(InInt)$
$L_8.$	$\vdash closed(\mathbb{Z}_5, \lambda xy.\bar{3}_5)$	$(ApplyAss\ ClosedConst\ L_3)$
$L_9.$	$\vdash closed(\mathbb{Z}_5, \lambda xy.x\bar{*}y)$	$(ReduceClosed\ ClComp\bar{*}\ L_4\ L_5\ L_6)$
$L_{10}.$	$\vdash closed(\mathbb{Z}_5, \lambda xy.(x\bar{*}y)\bar{+}\bar{3}_5)$	$(ReduceClosed\ ClComp\bar{+}\ L_7\ L_8\ L_9)$

Figure 9.3: Proof constructed by the ReduceToSpecial strategy.

1. Each residue class set RS_n is closed with respect to the operations: $\lambda xy.c$ if $c \in RS_n$ (corresponding to the theorem *ClosedConst*), $\lambda xy.x$ (*ClosedFV*), and $\lambda xy.y$ (*ClosedSV*).
2. Each complete residue class set $\mathbb{Z}_n$, which is closed under the binary operations op_1 and op_2, is also closed under the composed binary operation $\lambda xy.(x\ op_1\ y) \circ (x\ op_2\ y)$ where $\circ \in \{\bar{+}, \bar{-}, \bar{*}\}$ (corresponding to the theorems $ClComp\bar{+}$, $ClComp\bar{-}$, $ClComp\bar{*}$).

While the theorems under 1. can be applied by ApplyAss-B, it fails for the theorems under 2. This is due to the fact that the necessary instantiations for the operations op_1 and op_2 cannot be found by first-order matching. However, the algorithm of the ReduceClosed-B method can decide whether the theorem is applicable. For instance, when applying the theorem $ClComp\bar{+}$

$$\forall n : \mathbb{Z}.\forall op_1.\forall op_2.(closed(\mathbb{Z}_n, op_1) \wedge closed(\mathbb{Z}_n, op_2)) \Rightarrow$$
$$closed(\mathbb{Z}_n, \lambda x, y.(x\ op_1\ y)\bar{+}(x\ op_2\ y))$$

to line L_{10} in Figure 9.3, ReduceClosed-B computes the necessary instantiations for the operations op_1 and op_2, namely $\lambda xy.x\bar{*}y$ and $\lambda xy.\bar{3}_5$. Like applications of ApplyAss-B, also applications of ReduceClosed-B introduce the premises of the applied theorem as new goals (here L_7, L_8, L_9), which have to be tackled subsequently.

Like the EquSolve strategy, ReduceToSpecial is independent of the size of the residue class set. Theoretically, it is applicable to all types of problems in our domain. Whether it succeeds on a given problem depends on whether suitable theorems are available in the knowledge base.

We have experimented with bookkeeping already solved problems and trying to reduce new problems to these. However, this is not feasible since for large sets of problems the comparison of a new problem with those already solved is rather expensive.

9.1.4 Treating Direct Products

So far, we have explained the strategies with residue class structures with simple sets. The strategies are also able to handle direct products of residue class structures. In the following, we first introduce the necessary notions used in ΩMega to formalize direct products of structures. Afterwards, we explain with an example how the introduced strategies deal with direct products of structures.

Formally, we define direct products of residue class sets via iterated pairing of arbitrary residue class sets. Operations on direct products are pairs of the operations on the components of the direct products. First, we define the notion of pairs of elements with the following *pairing function*:

$$Pair \equiv \lambda x_\alpha.\lambda y_\beta.\lambda g_{\alpha\beta o}.g(x, y)$$

In order to access the 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 ANDREWS 's book [7] on page 185 .

$$\begin{aligned} LProj &\equiv \lambda p_{(\alpha\beta o)o}\textbf{.}\,\iota x_{\alpha}\textbf{.}\,\exists y_{\beta}\textbf{.}\,p\dot{=}Pair(x,y) \\ RProj &\equiv \lambda p_{(\alpha\beta o)o}\textbf{.}\,\iota y_{\beta}\textbf{.}\,\exists x_{\alpha}\textbf{.}\,p\dot{=}Pair(x,y) \end{aligned}$$

Next, we define the direct product of two sets as the set of all pairs of elements of the respective sets; that is:

$$\otimes \;\equiv\; \lambda U_{\alpha o}\textbf{.}\,\lambda V_{\beta o}\textbf{.}\,\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\textbf{.}\,[LProj(p) \in U] \wedge [RProj(p) \in V].$$

Finally, we define operations on direct products as pairs of the operations of the components of the direct product:

$$\begin{aligned} \times \;\equiv\; & \lambda U_{\alpha o}\textbf{.}\,\lambda V_{\beta o}\textbf{.}\,\lambda \circ^1_{\alpha\alpha\alpha}\textbf{.}\,\lambda \circ^2_{\beta\beta\beta}\textbf{.}\,\lambda p_{(\alpha\beta)((\alpha\beta o)o)}\textbf{.}\,\lambda q_{(\alpha\beta)((\alpha\beta o)o)}\textbf{.} \\ & \quad Pair(LProj(p) \circ^1 LProj(q), RProj(p) \circ^2 RProj(q)). \end{aligned}$$

Notation 9.1: In the remainder, we denote pairs of operations as $(\circ^1{\times}\circ^2)$. Moreover, we write direct products of sets as $U_1{\otimes}U_2$.

In case the given set is a direct product of residue class sets and the given operation is an operation on such a direct product of sets, then the proofs constructed by the EquSolve and the TryAndError strategy are only 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 9.1.1 and 9.1.2. We do not repeat all the details of these proofs and just describe the differences. The existential quantification

$$\exists z{:}_{\mathbb{Z}_2 \otimes \mathbb{Z}_2}\textbf{.}\,(cl_2(c_1), cl_2(c_2))\ [\bar{+}{\times}\bar{*}]\ z\dot{=}(\bar{0}_2, \bar{0}_2)$$

is rewritten to

$$\exists x{:}_{\mathbb{Z}_2}\textbf{.}\,\exists y{:}_{\mathbb{Z}_2}\textbf{.}\,(cl_2(c_1), cl_2(c_2))\ [\bar{+}{\times}\bar{*}]\ (x, y)\dot{=}(\bar{0}_2, \bar{0}_2),$$

to which $\exists$`IResclass` is applied twice. The resulting equation on tuples

$$(cl_2(c_1), cl_2(c_2))\ [\bar{+}{\times}\bar{*}]\ (cl_2(mv_1), cl_2(mv_2))\dot{=}(\bar{0}_2, \bar{0}_2)$$

is split into equations on the components

$$cl_2(c_1)\bar{+}cl_2(mv_1)\dot{=}\bar{0}_2 \quad \wedge \quad cl_2(c_2)\bar{*}cl_2(mv_2)\dot{=}\bar{0}_2.$$

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 performed by methods that are included in the strategies EquSolve and TryAndError.

9.1.5 Automatically Classifying Residue Class Structures

For a given residue class structure we can stepwise prove properties in order to classify the given structure in terms of the algebraic structure it forms. 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.

We implemented the automatic exploration of properties in a module in ΩMEGA, which we call the exploration module. In the sequel, we explain how this module works.

9.1.5.1 Classifying Structures with One Operation

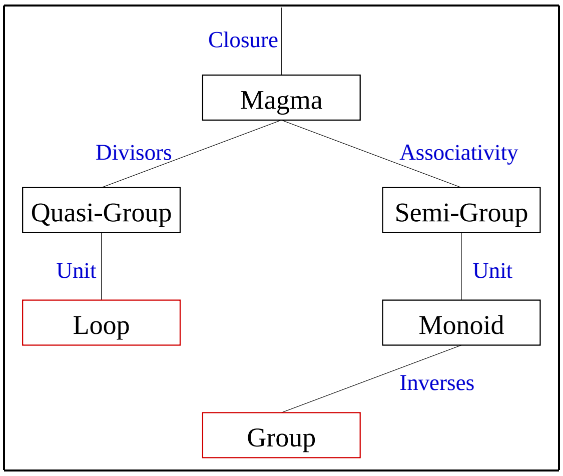


Figure 9.4: Classification schema for sets with one operation.

The main idea of the classification of residue class structures is to stepwise check properties of the structure in a schematic order. The results of these checks eventually gives an answer to the question what kind of algebraic entity the input structure forms. The classification schema for a residue class set together with a single operation is displayed in figure 9.4.

First, the module checks whether the given structure is closed under the operation. In case it can be proved that the structure is not closed the classification stops at this point. Otherwise, the structure in question forms a magma. The classification proceeds along the right branch of the schema in Figure 9.4. 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. 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, it is a loop. If the structure forms a loop, the module does not have to check any further since the structure is not a group because the module checked already that it is non-associative. Once the classification with respect to the schema in Figure 9.4 is finished and the structure is at least a magma, it is always checked whether it is Abelian.

The check and the proof of a single property are done in three steps: First the likely answer to whether a certain property holds or not is computed using the computer algebra systems Maple and GAP or the model generator SEM. To perform the tests with Maple and GAP or SEM the exploration module uses functionalities similar to the functionalities employed by the ComputeInstbyCasAndMG instantiation strategy. 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 passed to Multi, which tries to discharge it immediately by constructing a proof plan as described in the previous sections. If the proof planning process fails, then the negated proof obligation is constructed and passed to Multi 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.

9.1.5.2 Classifying Structures with two Operations

So far, we were only concerned with the classification of residue class sets together with one binary operation. 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 9.1.5. If $(RS_n, \circ)$ is an Abelian group, we try to establish distributivity of $\star$ over $\circ$.

If distributivity can be 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 section. 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.

9.2 Proof Plans of Isomorphism Problems

In the last section, we explained how Multi creates proof plans for simple properties of residue classes and discussed the classification of residue class structures in terms of the algebraic entity they form. In this section, we shall examine how Multi creates proof plans for the problems that two given residue class structures are either isomorphic or not isomorphic to each other. We shall reuse the same, albeit slightly extended, strategies developed for simple properties and a new **PPlanner** strategy as well as new **InstMeta**, **ATP**, and **BackTrack** strategies.

For the simple properties, Multi did interleave **PPlanner** strategies only with **BackTrack** or **InstMeta** strategies but not with each other. For the construction of isomorphism or non-isomorphism proof plans Multi relies on the combination and interleaving of different **PPlanner** strategies. This cooperation is not realized via interrupts of one **PPlanner** strategy. Rather, when one **PPlanner** strategy fails, strategic control rules prefer the application of other **PPlanner** strategies to the failure subgoals instead of backtracking. We shall explain this *failure-driven cooperation* in more detail as we go along and illustrate it with examples.

As for simple properties the strategic control specifies also for isomorphism or non-isomorphism problems (as well as for subproblems such as to show injectivity, surjectivity, or homomorphy) that the strategies ReduceToSpecial, EquSolve, and TryAndError are always tested in this order.

The exploration presented in section 9.1.5 returns sets of magmas, Abelian magmas, semi-groups, etc. This, however, does not indicate whether these structures are actually different (i.e., not isomorphic to each other) or just different representations of the same structure. The proof techniques we present in this chapter enable the further classification of residue class structures by dividing them into isomorphism classes.

This section is structured as follows: We first describe how both isomorphism and non-isomorphism proofs are planned. Afterwards, we point out the peculiarities when residue class structures with direct products are involved. Finally, we present the extensions of the exploration module to automatically classify residue class structures into isomorphism classes.

9.2.1 Isomorphism Proofs

Multi employs the same strategies already described in section 9.1 with the same methods that were already needed to prove simple properties of residue class sets. We added only two methods for the introduction of isomorphism mappings to the TryAndError and EquSolve strategies. Contrary to the proofs in section 9.1 that could be solved in most cases within one strategy, for isomorphism proofs different strategies have to cooperate to construct a proof plan. This means that Multi switches from the strategy EquSolve to either TryAndError or ReduceToSpecial.

9.2.1.1 Using the TryAndError Strategy

For the proof that two given structures are isomorphic, a mapping has to be constructed that is a bijective homomorphism from the one structure to the other structure. 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 explicitly specified as an element of the codomain. Following the ideas described already in section 9.1.1, the strategy TryAndError performs a case analysis for the different possibilities for defining the mapping. If TryAndError fails to prove bijectivity or the homomorphism property for a mapping, then it constructs — after backtracking — the next mapping and tries to prove bijectivity and the homomorphism properties.

We illustrate this with the problem that $(\mathbb{Z}_2, \bar{+})$ is isomorphic to $(\mathbb{Z}_3 \backslash \{\bar{0}\}, \bar{*})$. Figure 9.5 displays a part of the $\mathcal{PDS}$ for this problem.

The topmost case-split (i.e., the case-split over the possible instantiations of the isomorphism mapping) is introduced with the application of the ∃IResclFunc-B method in line L_{98}. ∃IResclFunc-B introduces a constant h' for the existentially quantified variable h, which denotes a function from $\mathbb{Z}_2$ to $\mathbb{Z}_3 \setminus \{0\}$. This function is also explicitly introduced in line L_1 as the formalization of a pointwise function

$$h' : \mathbb{Z}_2 \longrightarrow \mathbb{Z}_3 \setminus \{\bar{0}_3\} \quad \text{with} \quad h'(x) \doteq \begin{cases} cl_3(mv_1), & \text{if } x \doteq \bar{0}_2 \\ cl_3(mv_2), & \text{if } x \doteq \bar{1}_2 \end{cases},$$

where the mv_i are meta-variables that can be instantiated by elements of the range, i.e., by 1 or 2 in our example (see L_{96}). Then, TryAndError searches in the usual way (see section 9.1.1) for an appropriate combination of mv_1 and mv_2 that yields a function h', for which TryAndError can show the homomorphism property and bijectivity of h' in line L_{97}.

L_1.	L_1	$\vdash h' \doteq \lambda x_\centerdot (that\ y_\centerdot (x \doteq \bar{0}_2 \Rightarrow y \doteq cl_3(mv_1)) \wedge$ $(x \doteq \bar{1}_2 \Rightarrow y \doteq cl_3(mv_2)))$	(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}	$\vdash c_1 \doteq 0$	(Hyp)
L_{11}.	L_{11}	$\vdash c_2 \doteq 1$	(Hyp)
		$\vdots$	
L_{70}.	$\mathcal{H}_3$	$\vdash 1 \neq 2$	($\neq$REFLEXONNUM-B)
L_{71}.	$\mathcal{H}_3$	$\vdash 1 \neq 2 \vee 0 \doteq 1$	($\vee$IL-B L_{70})
L_{72}.	$\mathcal{H}_3$	$\vdash cl_3(1) \neq cl_3(2) \vee 0 \doteq 1$	(CONCONGCL-B L_{71})
L_{73}.	$\mathcal{H}_3$	$\vdash h'(\bar{0}_2) \neq h'(\bar{1}_2) \vee 0 \doteq 1$	(APPLYFUNCTION-B L_1 L_{72})
L_{74}.	$\mathcal{H}_3$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee c_1 \doteq c_2$	(SIMPLIFYNUM-B L_{10} L_{11} L_{73})
L_{75}.	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee c_1 \doteq c_2$	($\vee$E**-B L_5 $L_6 L_{74} \ldots$)
L_{76}.	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) \neq h'(cl_2(c_2)) \vee cl_2(c_1) \doteq cl_2(c_2)$	(CONCONGCL-B L_{75})
L_{77}.	$\mathcal{H}_2$	$\vdash h'(cl_2(c_1)) \doteq h'(cl_2(c_2)) \Rightarrow cl_2(c_1) \doteq cl_2(c_2)$	($\vee$2 $\Rightarrow$-B L_{76})
L_{78}.	$\mathcal{H}_1$	$\vdash \forall y{:}\mathbb{Z}_{2\centerdot} h'(cl_2(c_1)) \doteq h'(y) \Rightarrow cl_2(c_1) \doteq y$	($\forall$I-B L_{77})
L_{79}.	L_1	$\vdash \forall x{:}\mathbb{Z}_2, y{:}\mathbb{Z}_{2\centerdot} h'(x) \doteq h'(y) \Rightarrow x \doteq y$	($\forall$I-B L_{78})
L_{80}.	L_1	$\vdash Inj(h', \mathbb{Z}_2)$	(DEFNUNFOLD-B L_{79})
		$\vdots$	
L_{96}.	L_1	$\vdash mv_1 \in \{1,2\} \wedge mv_2 \in \{1,2\}$	($\wedge$I-B $\ldots$)
L_{97}.	L_1	$\vdash (Inj(h', \mathbb{Z}_2) \wedge Surj(h', \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\}) \wedge$ $Hom(h', \mathbb{Z}_2, \lambda xy_\centerdot x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy_\centerdot x \bar{*} y))$	($\wedge$I-B $\ldots$)
L_{98}.		$\vdash \exists h_\centerdot (Inj(h, \mathbb{Z}_2) \wedge Surj(h, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\}) \wedge$ $Hom(h, \mathbb{Z}_2, \lambda xy_\centerdot x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy_\centerdot x \bar{*} y))$	($\exists$IRESCLFUNC-B L_{96} L_{97})
L_{99}.		$\vdash Iso(\mathbb{Z}_2, \lambda xy_\centerdot x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy_\centerdot x \bar{*} y)$	(DEFNUNFOLD-B L_{98})

$\mathcal{H}_1 = \{L_1, L_5\}$, $\mathcal{H}_2 = \{L_1, L_5, L_6\}$, $\mathcal{H}_3 = \{L_1, L_5, L_6, L_{10}, L_{11}\}$

Figure 9.5: Introduction of the pointwise defined function.

In order to shortcut the search for the right function h' we extended the **INSTMETA** strategy ComputeInstbyCasAndMG such that it can provide instantiations for meta-variables, which are part of the pointwise function specification. ComputeInstbyCasAndMG can either employ the computer algebra system MAPLE or the model generator SEM to obtain an isomorphism between the structures $(RS^1_n, \circ_1)$ and $(RS^2_m, \circ_2)$. When employing MAPLE, ComputeInstbyCasAndMG asks MAPLE 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 of equations is generated by the instantiations of 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 $cl_n(i), cl_n(j) \in RS^1_n$. Thus, $h(cl_n(l))$ is substituted by x_l in our equation system. When MAPLE returns a solution for the equation system in which the variables equal to elements of the integer set corresponding to RS^2_m, then the solution is a homomorphism between the structures. When there is a disjoint solution with $x_i \neq x_j$, for all $i \neq j$, then the solution is an isomorphism. When employing SEM, ComputeInstbyCasAndMG passes the multiplication tables of $(RS^1_n, \circ_1)$ and $(RS^2_m, \circ_2)$ to SEM and asks SEM to compute a model for a bijective function h, which satisfies the homomorphism equation.[4]

In the example in Figure 9.5 ComputeInstbyCasAndMG asks MAPLE 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. MAPLE returns $\{x_1 = 0, x_0 = 0\}$, $\{x_1 = 2, x_0 = 1\}$, $\{x_0 = 1, x_1 = 1\}$. ComputeInstbyCasAndMG analyzes the solutions and accepts the second one because it is a disjoint solution and all elements are in the codomain. Therefore,

[4] The fact that h should be bijective does not have to be formalized by logic formulas but can be specified as side condition on h in the input language of SEM.

ComputeInstbyCasAndMG adds the bindings $mv_1{:=}^b 1, mv_2{:=}^b 2$. The introduction of these bindings changes the function h' in line L_1 to the function $h'(\bar{0}_2)\dot{=}\bar{1}_3$, $h'(\bar{1}_2)\dot{=}\bar{2}_3$.

Beginning in line L_{80}, Figure 9.5 shows how the function h' is used during the proof planning process in the subproof for injectivity. The proof up to L_{73} results from the 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 the application of the ApplyFunction-B 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}.

For a given function h' Multi has to construct subproofs of n^2 cases for the properties injectivity, surjectivity, and homomorphy, respectively. Here, n is the cardinality of the structures. However, if no suitable instantiation can be computed, there are n^n pointwise defined functions to check, which becomes infeasible already for relatively small n.

9.2.1.2 Using the EquSolve Strategy

During the isomorphism proof we have to show injectivity, surjectivity, and the homomorphism property for the introduced mapping. To construct proofs for these properties by a complete case analysis as performed by TryAndError can become quite lengthy. In order to tackle isomorphism problems with the EquSolve strategy we need a more compact form to represent the isomorphism function, namely 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. The subproof for injectivity has to show that for any two distinct elements the images differ; this cannot be done with the EquSolve strategy.

We added the functionality for the construction of the interpolation polynomial to the **InstMeta** strategy ComputeInstbyCasAndMG. ComputeInstbyCasAndMG employs either Maple or SEM to compute a pointwise defined function as described in the previous section. Then, it employs Maple to compute a polynomial that interpolates the pointwise function. ComputeInstbyCasAndMG does not use a standard algorithm for interpolating sparse polynomials (see for example [257, 258, 254]) as these do not necessarily return the best possible interpolation polynomial for our purpose. Moreover, some of the algorithms, for instance in Maple, are not sufficient for our purposes.[5] This is especially true for the case of multi-variate polynomial interpolation that is necessary for dealing with residue class sets that are composed of direct products, which we will describe in more detail in section 9.2.3. Thus, we have decided to implement a simple search algorithm in ComputeInstbyCasAndMG to find a suitable interpolation polynomial of minimal degree. This is feasible as ComputeInstbyCasAndMG has to handle only relatively small mappings.

In detail, the interpolation proceeds as follows: Given a pointwise defined isomorphism function $h{:}cl_n(x_i){\in}RS^1_n \to cl_m(y_i){\in}RS^2_m$ ComputeInstbyCasAndMG asks Maple to solve the system 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. When Maple returns a solution for $a_d, \ldots, a_0$, we have found an interpolating polynomial. If there is no solution, a polynomial with degree $d+1$ will be sent to Maple. This procedure terminates latest when $d = m - 1$.

[5] Maple's algorithms `interp` and `Interp` cannot always handle the interpolation of functions where a non-prime modulo factor is involved.

		$\vdots$	
$L_{50}.$		$\vdash Inj(mv_h, \mathbb{Z}_2)$	$(\ldots)$
		$\vdots$	
$L_{60}.$	L_{60}	$\vdash cl_2(c) \in \mathbb{Z}_2$	(Hyp)
		$\vdots$	
$L_{75}.$	L_{60}	$\vdash (mv_y + 1)\ mod\ 2 \doteq c\ mod\ 2_{\{mv_y :=^b c-1\}}$	(SolveEquation-b)
$L_{76}.$	L_{60}	$\vdash cl_2(mv_y) \bar{+} \bar{1}_2 \doteq cl_2(c)$	(ConCongCl-B L_{75})
$L_{77}.$	L_{60}	$\vdash mv_y \in \{0, 1\}$	$(Open)$
$L_{78}.$	L_{60}	$\vdash \exists y{:}\mathbb{Z}_2 \bullet y \bar{+} \bar{1}_2 \doteq c$	$(\exists I Resclass\ L_{76}\ L_{77})$
$L_{79}.$		$\vdash \forall x{:}\mathbb{Z}_2 \bullet \exists y{:}\mathbb{Z}_2 \bullet y \bar{+} \bar{1}_2 \doteq x$	($\forall$IRescalss-B L_{78})
$L_{80}.$		$\vdash Surj(\lambda x \bullet x \bar{+} \bar{1}_2, \mathbb{Z}_2, \mathbb{Z}_2)$	(DefnUnfold-B L_{79})
$L_{81}.$		$\vdash Inj(mv_h, \mathbb{Z}_2) \wedge Surj(mv_h, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\})$	($\wedge$I-B L_{80} L_{50})
		$\vdots$	
$L_{96}.$		$\vdash Hom(mv_h, \mathbb{Z}_2, \lambda xy \bullet x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy \bullet x \bar{*} y))$	(DefnUnfold-B L_{95})
$L_{97}.$		$\vdash (Inj(mv_h, \mathbb{Z}_2) \wedge Surj(mv_h, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\}) \wedge$ $Hom(mv_h, \mathbb{Z}_2, \lambda xy \bullet x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy \bullet x \bar{*} y))$	($\wedge$I-B L_{96} L_{81})
$L_{98}.$		$\vdash \exists h \bullet (Inj(h, \mathbb{Z}_2) \wedge Surj(h, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\}) \wedge$ $Hom(h, \mathbb{Z}_2, \lambda xy \bullet x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy \bullet x \bar{*} y))$	($\exists$I-B L_{97})
$L_{99}.$		$\vdash Iso(\mathbb{Z}_2, \lambda xy \bullet x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy \bullet x \bar{*} y)$	(DefnUnfold-B L_{98})

Figure 9.6: Introduction of the interpolated function.

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 9.6. First, EquSolve expands the defined concept Iso in L_{99} and then introduces a meta-variable mv_h in line L_{97} for the existentially quantified variable h in L_{98}. For this meta-variable ComputeInstbyCasAndMG is applicable and Multi switches from EquSolve to ComputeInstbyCasAndMG. As in TryAndError (see section 9.1.1) the switch from EquSolve to ComputeInstbyCasAndMG and back is organized by the control rule `interrupt-if-inst-from-cas-or-mg`, which interrupts EquSolve and poses a demand for ComputeInstbyCasAndMG. ComputeInstbyCasAndMG finds the interpolation polynomial $x \to x + 1\ mod\ 2$ and adds the binding $mv_h :=^b \lambda x \bullet x \bar{+} \bar{1}_2$. This changes the line L_{97} to

$$\begin{aligned}&(Inj(\lambda x \bullet x \bar{+} \bar{1}_2, \mathbb{Z}_2) \wedge Surj(\lambda x \bullet x \bar{+} \bar{1}_2, \mathbb{Z}_2, \mathbb{Z}_3 \backslash \{\bar{0}_3\}) \wedge \\ &\qquad Hom(\lambda x \bullet x \bar{+} \bar{1}_2, \mathbb{Z}_2, \lambda xy \bullet x \bar{+} y, \mathbb{Z}_3 \backslash \{\bar{0}_3\}, \lambda xy \bullet x \bar{*} y))\end{aligned}$$

Then, EquSolve has to show the properties of injectivity, homomorphy, and surjectivity for this interpolation polynomial. In Figure 9.6 we have carried out only the details for the subproof of surjectivity, in which the problem is reduced to an equation over integers that can be solved by Maple employing the SolveEquation-b method similar to the proof in section 9.1.2. The proof of the homomorphism property works analogously. The proof for injectivity in L_{50}, however, cannot be constructed with the EquSolve strategy for the reasons explained above. Thus, when EquSolve fails to construct a proof for L_{50}, then Multi should not perform backtracking with respect to the task with goal L_{50} but should prefer other strategies, which can deal with this line-task, in particular, TryAndError or ReduceToSpecial. This is realized by the strategic control rule `preferotherjob-if-EquSolvefailure`, which states that if EquSolve fails on particular line-tasks and there are job offers of TryAndError or ReduceToSpecial for these line-tasks, then these job offers are preferred before job offers of **BackTrack** strategies.[6] When EquSolve fails to prove the surjectivity or homomorphy subgoals, then Multi has to deal with those subproblems again at the strategic level. Guided by the described strategic control

[6] `preferotherjob-if-EquSolvefailure` has a higher priority as the strategic control rule `prefer-backtrack-if-failure` introduced in section 6.2.3. Hence, it "overwrites" `prefer-backtrack-if-failure`.

rule Multi would then prefer to try first TryAndError or ReduceToSpecial on the subgoals before backtracking. How the strategy ReduceToSpecial is applied in this context is described in the next section. In case the TryAndError strategy is applied, the case analysis is conducted with the interpolation polynomial instead with the pointwise function as in section 9.2.1.

As opposed to TryAndError, which can find an isomorphism by search, EquSolve can succeed only, if ComputeInstbyCasAndMG can provide an interpolation polynom. Thus, the success of EquSolve depends on the capabilities of Maple.

9.2.1.3 Using the ReduceToSpecial Strategy

Since ΩMEGA's database does not contain theorems on isomorphism problems, ReduceToSpecial is not applicable to the original theorem, but it comes into play, when a subgoal, in particular an injectivity subgoal, has to be proved. Here, we can exploit the following simple mathematical fact:

> *A surjective mapping between two finite sets with the same cardinality is injective.*

The proof of injectivity becomes simply a theorem application, if Multi can prove by other means (i.e., EquSolve) that a given mapping is surjective. 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 let ReduceToSpecial finish the proof.

9.2.2 Non-Isomorphism Problems

In this section, we shall discuss how Multi can construct proof plans for non-isomorphism problems. If the two structures involved are of different cardinalities, 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 but concentrate instead on the more interesting cases. For tackling non-isomorphism problems we implemented the following three proof techniques:

1. Show that each possible mapping between the two structures is not isomorphic. This is an exhaustive case analysis for which we employ the slightly extended TryAndError strategy.
2. Isomorphic structures have all algebraic properties in common. 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. This technique is realized by interleaving the (slightly extended) EquSolve strategy with the **ATP** strategy CallTramp and the **InstMeta** strategy ComputeInstbyHR, which employs HR [58] a system for theory formation to obtain a property that holds for one structure but not for the other.
3. We construct a contradiction by assuming there exists an isomorphism between the two residue class structures and deriving that it is not injective. For this technique we have implemented a new strategy, called NotInjNotIso.

Also on non-isomorphism problems the strategic control among the strategies ReduceToSpecial, EquSolve, and TryAndError stays the same: they are tried in this order. The new strategy NotInjNotIso is tried after EquSolve and before TryAndError.

9.2.2.1 Using the TryAndError Strategy

As already stated in section 9.1.1, the two basic principles of the TryAndError strategy are to tackle quantified statements 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 one residue class set into the other that it is either not injective, not surjective, or not a homomorphism.

L_1.	L_1	$\vdash h' \doteq \lambda x. (that\ y. (x \doteq \bar{0}_4 \Rightarrow y \doteq cl_4(c_1)) \wedge$ $(x \doteq \bar{1}_4 \Rightarrow y \doteq cl_4(c_2)) \wedge$ $(x \doteq \bar{2}_4 \Rightarrow y \doteq cl_4(c_3)) \wedge$ $(x \doteq \bar{3}_4 \Rightarrow y \doteq cl_4(c_4)))$	(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	$\vdash c_1 \doteq 0$	(Hyp)
L_7.	L_7	$\vdash c_2 \doteq 0$	(Hyp)
L_8.	L_8	$\vdash c_3 \doteq 0$	(Hyp)
L_9.	L_9	$\vdash c_4 \doteq 0$	(Hyp)
L_{10}.	L_{10}	$\vdash c_1 \doteq 1$	(Hyp)
		$\vdots$	
L_{75}.	$\mathcal{H}_3$	$\vdash (\neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4))$	($\vee$IR-B L_{74})
		$\vdots$	
L_{95}.	$\mathcal{H}_2$	$\vdash (\neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4))$	($\vee$IL-B L_{94})
L_{96}.	$\mathcal{H}_1$	$\vdash (\neg Inj(h', \mathbb{Z}_4) \vee \neg Surj(h', \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h', \mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4))$	($\vee$E**-B L_2 L_3 L_4 L_5 L_{95} L_{75} ...)
L_{97}.		$\vdash \forall h{:}F(\mathbb{Z}_4, \mathbb{Z}_4).$ $(\neg Inj(h, \mathbb{Z}_4) \vee \neg Surj(h, \mathbb{Z}_4, \mathbb{Z}_4) \vee$ $\neg Hom(h, \mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4))$	($\forall$IResclFunc-B L_{96})
L_{98}.		$\vdash \neg \exists h{:}F(\mathbb{Z}_4, \mathbb{Z}_4).$ $(Inj(h, \mathbb{Z}_4) \wedge Surj(h, \mathbb{Z}_4, \mathbb{Z}_4) \wedge$ $Hom(h, \mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4))$	(PullNeg-B L_{97})
L_{99}.		$\vdash \neg Iso(\mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4, \mathbb{Z}_4, \lambda xy. \bar{2}_4)$	(DefnUnfold-B L_{98})
$\mathcal{H}_1 = \{L_1, L_2, L_3, L_4, L_5\}$, $\mathcal{H}_2 = \mathcal{H}_1 \cup \{L_6, L_7, L_8, L_9\}$, $\mathcal{H}_3 = \mathcal{H}_1 \cup \{L_7, L_8, L_9, L_{10}\}$			

Figure 9.7: Proof constructed by the TryAndError strategy.

Figure 9.7 displays a segment of the $\mathcal{PDS}$ for the non-isomorphism problem that the two Abelian semi-groups $(\mathbb{Z}_4, \lambda xy. x \bar{*} y \bar{*} \bar{2}_4)$ and $(\mathbb{Z}_4, \lambda xy. \bar{2}_4)$ are not isomorphic constructed by TryAndError.[7] The proof works in the following way: after unfolding the definition of isomorphism in line L_{99}, the application of the method PullNeg-B pushes the negation to the inner-most formulas. Next, TryAndError applies $\forall$IResclFunc-B, a method for the elimination of universally quantified goals that is the dual of the $\exists$IResclFunc-B method introduced in section 9.2.1. $\forall$IResclFunc-B instantiates the variable h for a mapping between the two given residue class sets with a constant h' and 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 the case-split over all possible mappings between the residue class sets, i.e., all possible combinations of constants c_1 to c_4. It is introduced by the application of $\vee$E**-B to line L_{96} with respect to the lines L_2 through L_5. The case-split leads to 256 new open subgoals of which we depict only two, i.e., lines L_{95} and L_{75}, in

[7] We have renumbered the lines in order to preserve space.

Figure 9.7. Likewise, we depict 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 has a different combination of these constants in its hypotheses. It 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, TryAndError can show 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 it has to consider n^n different possible functions. Thus, in practice this strategy is not feasible for structures of cardinality larger than four.

9.2.2.2 Using Discriminants

If two structures are isomorphic, they have all algebraic properties in common. Thus, in order to show that two structures are not isomorphic, it suffices to show that one property holds for one structure but not for the other. Such a property is called a *discriminant* for the two structures.

For example, consider the pairwise non-isomorphic quasi-groups S^1, S^2, S^3 depicted with their respective multiplication tables in Figure 9.8. When comparing the tables of S^1 and S^2, one discriminant is fairly obvious: while S^1 has only $\bar{0}_5$ on the main diagonal, all elements on the main diagonal of S^2 are distinct. Thus, the property we can use is $\exists x_\bullet \forall y_\bullet x \doteq y \circ y$. Things become less obvious for the multiplication tables of S^2 and S^3. Here, one property of S^3, which does not hold for S^2, is $\forall x_\bullet \forall y_\bullet (x \circ x \doteq y) \Rightarrow (y \circ y \doteq x)$.

$S^1 \doteq (\mathbb{Z}_5, \bar{-})$

S^1	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$
$\bar{0}_5$	$\bar{0}_5$	$\bar{4}_5$	$\bar{3}_5$	$\bar{2}_5$	$\bar{1}_5$
$\bar{1}_5$	$\bar{1}_5$	$\bar{0}_5$	$\bar{4}_5$	$\bar{3}_5$	$\bar{2}_5$
$\bar{2}_5$	$\bar{2}_5$	$\bar{1}_5$	$\bar{0}_5$	$\bar{4}_5$	$\bar{3}_5$
$\bar{3}_5$	$\bar{3}_5$	$\bar{2}_5$	$\bar{1}_5$	$\bar{0}_5$	$\bar{4}_5$
$\bar{4}_5$	$\bar{4}_5$	$\bar{3}_5$	$\bar{2}_5$	$\bar{1}_5$	$\bar{0}_5$

$S^2 \doteq (\mathbb{Z}_5, \lambda xy_\bullet (\bar{2}_5 \bar{*} x) \bar{+} y)$

S^2	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$
$\bar{0}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$
$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$
$\bar{2}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$
$\bar{3}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$
$\bar{4}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$

$S^3 \doteq (\mathbb{Z}_5, \lambda xy_\bullet (\bar{3}_5 \bar{*} x) \bar{+} y)$

S^3	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$
$\bar{0}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$
$\bar{1}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$
$\bar{2}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$
$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$	$\bar{2}_5$	$\bar{3}_5$
$\bar{4}_5$	$\bar{2}_5$	$\bar{3}_5$	$\bar{4}_5$	$\bar{0}_5$	$\bar{1}_5$

Figure 9.8: Some quasi-group multiplication tables.

The generalized proof procedure is as follows: given two structures S^1 and S^2 we have to:

1. find a discriminant P,
2. show that $P(S^1)$ holds,
3. show that $\neg P(S^2)$ holds, and
4. show that $\forall X_\bullet \forall Y_\bullet P(X) \wedge \neg P(Y) \Rightarrow X \not\simeq Y$ holds (where X and Y are variables for structures).[8]

The single proof parts combine to the following proof sketch:

[8] While step 4 is fairly obvious for a human mathematician, it is crucial for a formal proof.

$$\dfrac{\dfrac{\begin{matrix}?\\ \vdots\;(2)\\ P(S_1)\end{matrix}\qquad\begin{matrix}?\\ \vdots\;(3)\\ \neg P(S_2)\end{matrix}}{P(RS_1)\wedge\neg P(S_2)}\wedge_I\qquad\dfrac{\begin{matrix}?\\ \vdots\;(4)\\ \forall X\mathbf{.}\forall Y\mathbf{.}P(X)\wedge\neg P(Y)\Rightarrow X\not\simeq Y\end{matrix}}{P(S_1)\wedge\neg P(S_2)\Rightarrow S_1\not\simeq S_2}\forall_E(S_1,S_2)}{S_1\not\simeq S_2}\Rightarrow_E$$

The four problems 1 to 4 are solved by different strategies and different integrated systems. To compute a suitable discriminant P, we employ HR, a system for theory formation. The proofs that P is a discriminant for two given residue class structures (i.e., that $P(RS^1_n,\circ^1)$ and $\neg P(RS^2_m,\circ^2)$ holds) are done by **PPlanner** strategies. To obtain a formal proof that P is a discriminant for two arbitrary structures X and Y (i.e., step 4) we use first-order automated theorem provers.

We realized this technique as follows: we formalized the proof schema described above in the method IsoToDiscriminant-B, which we added to EquSolve.[9] The application of IsoToDiscriminant-B by EquSolve reduces the initial goal $\neg Iso(RS^1_n,\circ^1,RS^2_m,\circ^2)$ to three line-tasks with the goals

(1) $mv_P(RS^1_n,\circ^1)$,
(2) $\neg mv_P(RS^2_m,\circ^2)$, and
(3) $\forall Set^1,Op^1,Set^2,Op^2\mathbf{.}mv_P(Set^1,Op^1)\wedge\neg mv_P(Set^2,Op^2)\Rightarrow$
$[\neg Iso(Set^1,Op^1,Set^2,Op^2)]$

and an instantiation-task for the meta-variable mv_P, which substitutes the discriminant P.

Afterwards, EquSolve interrupts and poses demands to first apply the instantiation strategy ComputeInstbyHR to mv_P and then to apply the **ATP** strategy CallTramp (see section 6.2.4) to the goal (3). When both strategies succeed and EquSolve is re-invoked, then it tackles the remaining goals $P(RS^1_n,\circ^1)$ and $\neg P(RS^2_m,\circ^2)$, where the meta-variable mv_P is meanwhile bound to property P. $P(RS^1_n,\circ^1)$ and $\neg P(RS^2_m,\circ^2)$ are first tackled by EquSolve. If EquSolve fails to prove these subgoals[10], TryAndError is applied to them guided by the strategic control rule `preferotherjob-if-EquSolvefailure` that prefers job offers of other strategies for goals on which EquSolve fails (see section 9.2.1).

In the following, we illustrate the application of HR and the automated theorem provers with the problem that $\neg Iso(\mathbb{Z}_5,\bar{-},\mathbb{Z}_5,\lambda xy\mathbf{.}(\bar{2}_5\bar{*}x)\bar{+}y)$. HR offers as discriminant $\lambda Set\mathbf{.}\lambda Op\mathbf{.}\exists x{:}Set\mathbf{.}\forall y{:}Set\mathbf{.}\; x\doteq Op(y,y)$, which reduces the two goals for the **PPlanner** strategies to $\exists x{:}\mathbb{Z}_5\mathbf{.}\forall y{:}\mathbb{Z}_5\mathbf{.}x\doteq y\bar{-}y$ and $\neg\exists x{:}\mathbb{Z}_5\mathbf{.}\forall y{:}\mathbb{Z}_5\mathbf{.}x\doteq\bar{2}_5\bar{*}y)\bar{+}y$. Since these two goals are solved by the strategies EquSolve and TryAndError as usual we omit to further discuss them.

ComputeInstbyHR works similar to ComputeInstbyCasAndMG. When applied to an instantiation-task, it analyzes which kind of instantiation is needed and then applies HR to compute the actual instantiation. To obtain a discriminant ComputeInstbyHR uses HR's concept formation, which is achieved by using production rules that take one (or two) old concepts as input and output a new concept. The input for HR are the two structures for which a discriminant is needed and a set of production rules. In particular, we use the following four production rules of HR:

- `Compose:` composes functions using conjugation.

[9] We added IsoToDiscriminant-B to EquSolve since EquSolve is supposed to solve the goal $P(RS^1_n,\circ^1)$.

[10] Typically, EquSolve succeeds for $P(RS^1_n,\circ^1)$ and fails for $\neg P(RS^2_m,\circ^2)$.

- `Match:` equates variables in predicate definitions.
- `Forall:` introduces existential quantification.
- `Exists:` introduces universal quantification.

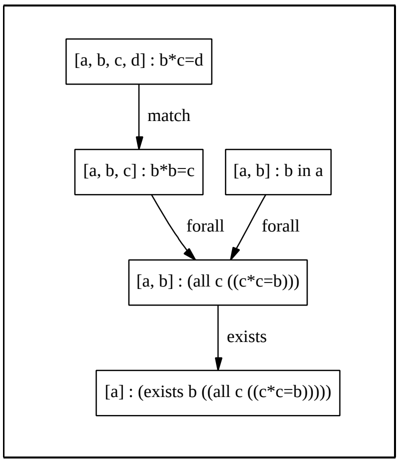


Figure 9.9: Example construction of HR.

As an example consider the concept of there being a single element on the diagonal of the multiplication table of an algebra, as is the case for $(\mathbb{Z}_5, \bar{-})$ but not for $(\mathbb{Z}_5, \lambda xy\text{.}(\bar{2}_5 \bar{*} x)\bar{+}y)$. This concept is constructed by HR using the match, forall and exists production rules, as depicted in Figure 9.9 from the basic concepts 'element of the algebra' and 'multiplication of two elements to give a third'. Using the match production rule with the multiplication concept, HR invents the notion of multiplying an element by itself. By using this in the forall production rule, it invents the concept of elements, which all other elements multiply by themselves to give. Then, using the exists production rule, HR invents the notion of algebras where there is such an element. The resulting property is expressed as an λ-term, which yields: $\lambda Set\text{.}\lambda Op\text{.}\exists x{:}Set\text{.}\forall y{:}Set\text{.}\ x \doteq Op(y,y)$. A more detailed discussion of the usage of HR by ComputeInstbyHR can be found in [167].

With respect to the binding $mv_P{:=}^b\,\lambda Set\text{.}\lambda Op\text{.}\exists x{:}Set\text{.}\forall y{:}Set\text{.}\ x \doteq Op(y,y)$ introduced by ComputeInstbyHR the goal (3) becomes:

$$\forall Set^1, Op^1, Set^2, Op^2\text{.}$$
$$[\exists x{:}Set^1\text{.}\forall y{:}Set^1\text{.}x \doteq Op^1(y,y)] \land \neg[\exists x{:}Set^2\text{.}\forall y{:}Set^2\text{.}x \doteq Op^2(y,y)]$$
$$\Rightarrow [(Set^1, Op^1) \not\sim (Set^2, Op^2)].$$

CallTramp succeeds to solve the goal, if one of the automated theorem provers interfaced by TRAMP succeeds.[11] TRAMP returns the corresponding ND-proof, which is stored for a potential expansion (see section 6.2.4). For our example, TRAMP produces ND-proofs containing between 71 (ND-proof transformed from SPASS proof) and 104 steps (from BLIKSEM proof).

We point out that the interface between MULTI and HR is currently not automated. Thus, currently the described technique does not work fully automatically. Rather, the instantiation strategy ComputeInstbyHR asks the user to supply HR's results interactively.

[11] The formula passed to TRAMP is a higher-order theorem since it contains quantifications on sets, operations, and the functions h and j. However, when TRAMP calls the connected automated theorem provers it creates a clause normal form of the problem and all the higher-order variables become constants (the theorem is negated for clause normalization).

9.2.2.3 Proof by Contradiction

In this section, we introduce the new strategy NotInjNotIso to tackle non-isomorphism problems. For the development of NotInjNotIso experiments with randomization and restarts techniques known from Artificial Intelligence were necessary, from which we acquired the control knowledge to guide the application of NotInjNotIso. Since these experiments were related only to the NotInjNotIso strategy and since their results are necessary to discuss the NotInjNotIso strategy, we shall describe them here and do not delay them to the general discussion of the conducted experiments in section 9.3.2.

The idea of NotInjNotIso is to construct an indirect proof that shows that two structures $(RS^1_{n_1}, \circ_1)$ and $(RS^2_{n_2}, \circ_2)$ are not isomorphic. The strategy first assumes that the two structures are isomorphic and that h is a bijective homomorphism from $(RS^1_{n_1}, \circ_1)$ to $(RS^2_{n_2}, \circ_2)$. If h is bijective, then it is also injective. The strategy then tries to find two elements $c_1, c_2 \in RS^1_{n_1}$ with $c_1 \neq c_2$ such that it can derive the equation $h(c_1) \doteq h(c_2)$. This contradicts the assumption of injectivity of h which implies that $h(c_1) \neq h(c_2)$ has to hold, if $c_1 \neq c_2$. Note that the proof works with respect to all possible homomorphisms h.

$L_1.$	$L_1 \vdash Iso(\mathbb{Z}_5, \lambda xy_{\bullet} x\bar{*}y, \mathbb{Z}_5, \lambda xy_{\bullet} x\bar{+}y)$	(Hyp)
	$\vdots$	
$L_6.$	$L_1 \vdash Inj(h, \mathbb{Z}_5)$	$(\wedge$E-F $\ldots)$
$L_7.$	$L_1 \vdash Hom(h, \mathbb{Z}_5, \lambda xy_{\bullet} x\bar{*}y, \mathbb{Z}_5, \lambda xy_{\bullet} x\bar{+}y)$	$(\wedge$E-F $\ldots)$
$L_8.$	$L_1 \vdash h(\bar{0}_5) \doteq h(\bar{0}_5)\bar{+}h(\bar{0}_5)$	(InstHomEqus-F L_7)
$L_9.$	$L_1 \vdash h(\bar{0}_5) \doteq h(\bar{0}_5)\bar{+}h(\bar{1}_5)$	(InstHomEqus-F L_7)
	$\vdots$	
$L_{88}.$	$L_1 \vdash ((((h(\bar{0}_5)\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{1}_5) \doteq h(\bar{1}_5)$	(SolveEquation-b)
$L_{89}.$	$L_1 \vdash (((h(\bar{0}_5)\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{1}_5) \doteq h(\bar{1}_5)$	(=Subst-B L_{88} L_8)
$L_{90}.$	$L_1 \vdash ((h(\bar{0}_5)\bar{+}h(\bar{0}_5))\bar{+}h(\bar{0}_5))\bar{+}h(\bar{1}_5) \doteq h(\bar{1}_5)$	(=Subst-B L_{89} L_8)
$L_{91}.$	$L_1 \vdash (h(\bar{0}_5)\bar{+}h(\bar{0}_5))\bar{+}h(\bar{1}_5) \doteq h(\bar{1}_5)$	(=Subst-B L_{90} L_8)
$L_{92}.$	$L_1 \vdash h(\bar{0}_5)\bar{+}h(\bar{1}_5) \doteq h(\bar{1}_5)$	(=Subst-B L_{91} L_8)
$L_{93}.$	$L_1 \vdash h(\bar{0}_5) \doteq h(\bar{1}_5)$	(=Subst-B L_{92} L_9)
	$\vdots$	
$L_{97}.$	$L_1 \vdash \neg Inj(h, \mathbb{Z}_5)$	$(\ldots)$
$L_{98}.$	$L_1 \vdash \bot$	$(\neg E\ L_{97}\ L_6)$
$L_{99}.$	$\vdash \neg Iso(\mathbb{Z}_5, \lambda xy_{\bullet} x\bar{*}y, \mathbb{Z}_5, \lambda xy_{\bullet} x\bar{+}y)$	(Contra-B L_{98})

Figure 9.10: Proof with the NotInjNotIso strategy.

Figure 9.10 shows a part of the proof with the NotInjNotIso strategy for the example problem $\neg Iso(\mathbb{Z}_5, \lambda xy_{\bullet} x\bar{*}y, \mathbb{Z}_5, \lambda xy_{\bullet} x\bar{+}y)$. The idea is to derive the contradiction in line L_{98} by assuming that there exists an isomorphism in line L_1. NotInjNotIso derives in the lines L_6 and L_7 the properties that all possible isomorphisms h have to be injective homomorphisms. Then, it derives from the homomorpshim property in L_7 the completely instantiated homomorphism equation system. In our example, this system consists of 25 single equations. In Figure 9.10 we show only two of these equations in the lines L_8 and L_9. The application of InstHomEqus-F introduces the simplified versions of the equations, which are of the general form $h(x \circ_1 y) \doteq h(x) \circ_2 h(y)$. The instantiation of the proper operations and the application to the arguments $x = \bar{0}_5$ and $y = \bar{0}_5$ results in the equation of line L_8 (similarly, the equation of line L_9 results from $x = \bar{0}_5$ and $y = \bar{1}_5$).

From the system of equations the NotInjNotIso strategy tries to derive that h is not injective. To prove this, it has to find two witnesses c_1 and c_2 for which $c_1 \neq c_2$ and $h(c_1) \doteq h(c_2)$ hold. In the proof in Figure 9.10 NotInjNotIso chooses $\bar{0}_5$ and $\bar{1}_5$

for c_1 and c_2, respectively. We omit the part of the proof that derives $\bar{0}_5 \neq \bar{1}_5$ and concentrate on the more difficult part to show $h(\bar{0}_5) \doteq h(\bar{1}_5)$ in line L_{93}. This goal is reduced to line L_{88} by successively applying equations from the equation system with the method =Subst-B. The formula of L_{88} is accepted by MAPLE as a generally valid equation (with respect to the modulo factor 5), and NotInjNotIso closes L_{88} by the method SOLVEEQUATION-B. Since line L_{97} contradicts the assumption of injectivity of h, MULTI can conclude the proof.

The essential part of an application of the NotInjNotIso strategy is the search for a sequence of applications of the =Subst-B method, which reduces $h(c_1) \doteq h(c_2)$ to an equation that can be shown by MAPLE. During this process NotInjNotIso has to make decisions about which instantiated homomorphism equation to apply next with the =Subst-B method. Since all instantiated homomorphism equations have the form $h(c) \doteq h(c_1) \circ h(c_2)$ the decision is, which subterm $h(\ldots)$ of the current goal to replace by a corresponding instantiated homomorphism equation. The idea to guide the selection is to prefer instantiated homomorphism equations whose application results in equations that contain as few as possible different $h(\ldots)$ expressions. Then, several occurrences of the same $h(\ldots)$ expression can be canceled (which is done by MAPLE) with respect to the modulo factor. For instance, in the final equation in line L_{88} in Figure 9.10 5 occurrences of $h(\bar{0})$ connected by $\bar{+}$ are canceled since $5 * h(\bar{0}_5)$ modulo 5 equals $\bar{0}_5$.

This idea is realized in the control rule `choose-next-equation`, which guides the decision for the next instantiated homomorphism equation by adopting the following heuristics:

(1) Prefer the application of an instantiated homomorphism equation that replaces in the current goal an occurrence of $h(c)$ such that $h(c)$ is the $h(\ldots)$ expression with the least occurrences in the goal.

(2) Among the remaining instantiated homomorphism equations prefer an equation that introduces the least number of $h(\ldots)$ expressions that are new in the goal.

We applied NotInjNotIso with this heuristic guidance to a testbed of 160 non-isomorphism problems over the residue class set $\mathbb{Z}_5$. Some example instances are:

1. $\neg Iso(\mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{*} y, \mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{+} y)$,
2. $\neg Iso(\mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{*} y, \mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{-} y)$,
3. $\neg Iso(\mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{+} y, \mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{-} y)$,
4. $\neg Iso(\mathbb{Z}_5, \lambda xy_{\blacksquare} x \bar{-} y, \mathbb{Z}_5, \lambda xy_{\blacksquare} \bar{2}_5 \bar{*} (x \bar{-} y))$.

The problem instances are constructed by combining structures of different algebraic categories (102 problems) and problems combining two quasi-group structures from different isomorphism classes (58 problems). For instance, problem 1 consists of a monoid structure and a group structure, problem 2 of a monoid structure and a quasi-group structure, problem 3 of a group and a quasi-group structure, and problem 4 of two quasi-group structures.

The application of NotInjNotIso to all problems of the testbed (we used a 2 hour time limit per proof attempt) revealed a surprisingly high variance in the performance of the strategy. On some of the problems it succeeded very fast (in the order of seconds) and produced short proof plans consisting only of a few applications of =Subst-B, whereas on other problems the planning process took much longer (in the order of several hundreds of seconds) and resulted in proof plans with many applications of =Subst-B. Furthermore, for over 30% of the instances no proof was found in 2 hours. Table 9.2 displays the performance extrema for these runs as well as the mean values over all successful runs. The values in brackets give the deviation from the mean.[12]

Figure 9.11 shows the underlying distribution of the run time for these experiments. We observe a large variance in run times for the various instances. In fact, the distribution exhibits *heavy-tailed* behavior [103, 105, 104], which is manifested in the long tail of the distribution stretching for several orders of magnitude.

Costs	Mean	Min.	Max.
Proof length	55	45 (18.2%)	83 (50.9%)
Run Time	483	8(98%)	7145(1380%)

Table 9.2: Statistics for successful runs (108 out of 160) on testbed using deterministic strategy.

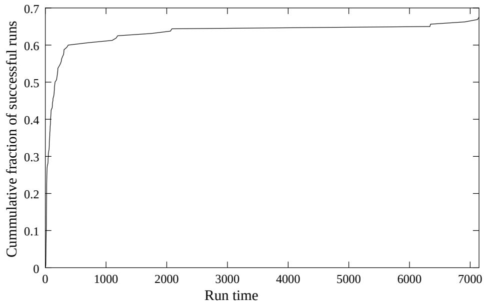


Figure 9.11: Run time distribution over testbed without randomization.

Gomes *et al.* have shown that one can take advantage of the large variations in run time of such heavy-tailed distributions by introducing an element of randomness into the search process, combined with a restart strategy. A key criterion for the success of such a randomization and restart approach is a large variance in different randomized runs with the same instance. To explore this issue, we considered multiple runs on a single instance by introducing a stochastic element into the planning process.

We extended `choose-next-equation` such that it randomly orders all instantiated homomorphism equations, which are ranked equally good. We ran this randomized version of NotInjNotIso 225 times on the following problem instance:

$$\neg Iso(\mathbb{Z}_5, (\bar{x}\bar{+}\bar{y})\bar{+}\bar{2}_5, \mathbb{Z}_5, (\bar{2}_5\bar{*}(\bar{x}\bar{+}\bar{y}))\bar{+}\bar{2}_5)$$

Interestingly, the run time distribution of the randomized NotInjNotIso strategy on the single instance also exhibits heavy-tailed behavior, see Figure 9.12. A detailed analysis is given in [160, 158]. This is an indication that the source of variance is inherent to the search process performed by NotInjNotIso.

Given that the heavy-tailedness is inherent in the search process, we can use a restart approach to improve the proof search performance. Figure 9.12 shows that the ascend of the cumulative cost distribution function is very steep at the beginning but becomes very flat beyond approximately 300 seconds. This steep ascend at the beginning indicates that there is a large fraction of short and successful runs whereas

[12] We measured search cost in terms of CPU time. Other measures appear less informative because of the hybrid nature of the proof planning process. For example, querying the external system Maple often takes a substantial fraction of the time; also, expression simplification rules can take significant time. Hence, CPU time appears to be a suitable overall performance measure.

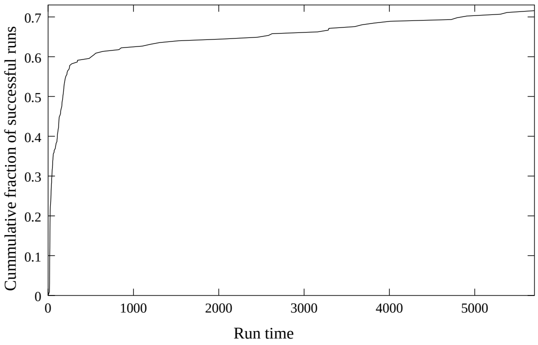


Figure 9.12: Run time distribution for single problem.

the flat ascend after 300 seconds provides evidence that the probability of finding a proof plan decreases considerably. Hence, it is advantageous to perform a sequence of restarts on a single instance (with a predefined cutoff) until reaching a successful run or the total time limit, instead of performing a single long run.

The cutoff and restart approach is captured in MULTI in two control rules. The interrupt control rule `interrupt-if-cutoff` in NotInjNotIso checks how much time NotInjNotIso did spend in a run so far. It interrupts NotInjNotIso, when the run time exceeds the predefined cutoff, and then poses a demand to backtrack the whole application of NotInjNotIso with the **BACKTRACK** strategy BackTrackPPlanner-Strategy. This strategy deletes complete **PPLANNER** actions comprising the deletion of all method-actions of the **PPLANNER** action as well as all actions that depend on these method-actions. When MULTI backtracks the application of NotInjNotIso, then the strategic control rule `reject-applied-offers` (see section 6.2.3) forbids to apply NotInjNotIso again to the same task (capturing the non-isomorphism problem). However, `reject-applied-offers` is overwritten by the strategic control rule `restart-`NotInjNotIso, which has a higher priority and allows to apply NotInjNotIso up to a predefined number of times.

Based on an analysis of the underlying distributions of the experiments for the full testbed and for the single problem we considered several cutoff and restart values, using a binary search strategy. The cutoff value of 80 seconds with 90 restarts provided the best results (see [158]). NotInjNotIso found proof plans for 156 of the 160 problems (97.5%) in an average time of 291.4 seconds (mean time of solved problems). Figure 9.13 plots the run time distribution of the resulting restart approach with cutoff 80 (log-log scale) on the problems of the testbed. The restart data is given by the curve that drops rapidly. The figure also shows the run time distribution of the deterministic strategy. The heavy-tailed nature of the run time distribution of the deterministic strategy is evident from the approximately linear behavior over several orders of magnitude of the tail of the distribution in the log-log plot. The sharp drop of the run time distribution of the restart strategy clearly indicates that this strategy does not exhibit heavy tailed behavior.

With respect to our results the cutoff value for non-isomorphism problems with $\mathbb{Z}_5$ in `interrupt-if-cutoff` is 80 seconds and `restart-`NotInjNotIso allows 90 restarts of NotInjNotIso on a non-isomorphism problem with $\mathbb{Z}_5$. We obtained analogous results on non-isomorphism problems of the residue class sets $\mathbb{Z}_2$, $\mathbb{Z}_3$, $\mathbb{Z}_4$, and $\mathbb{Z}_6$. The experiments conducted on these problem classes are described in [158]. There we report also experiments with randomization and restart approaches with the TryAndError strategy. The analysis of the underlying distributions did not exhibit heavy-tailed behavior.

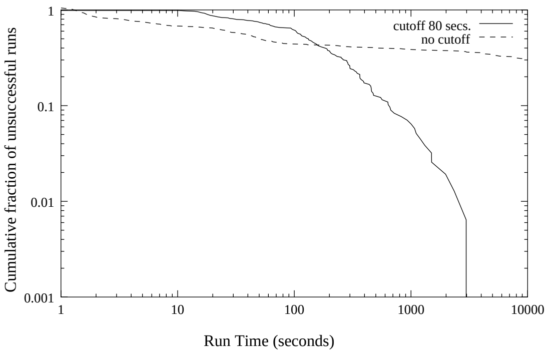


Figure 9.13: Log-Log plots of run time distribution over testbed with and without randomization.

9.2.3 Treating Direct Products

With minor extensions to our strategies the proving techniques for isomorphism problems and non-isomorphism problems in the residue class domain are also applicable to problems where the structures involved contain direct products of residue class sets. Apart from those methods already illustrated in section 9.1.4 that decompose quantifications and equations on tuples into the components, a few additions had to be made for tackling both isomorphism problems and non-isomorphism problems.

The pointwise defined function introduced by the TryAndError strategy for isomorphism problems maps in the case of direct products in the domain or codomain of the mapping, 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

$$h(x,y) \doteq \begin{cases} (mv_1, mv_2), \text{ if } (x,y) \doteq (c_1,c_1) \in RS^1_{n_1} \otimes RS^2_{n_2} \\ (mv_3, mv_4), \text{ if } (x,y) \doteq (c_1,c_2) \in RS^1_{n_1} \otimes RS^2_{n_2} \\ \quad \vdots \end{cases},$$

with $mv_1, mv_3, \ldots \in RS^3_{n_3}$ and $mv_2, mv_4, \ldots \in RS^4_{n_4}$. For non-isomorphism problems the codomain of the mapping contains constants instead of meta-variables.

Similarly, the interpolation polynom 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 there is one separate equation system for each component of the direct product in the codomain. Each equation system is of the form $h_i(x \circ_1 y) \doteq h_i(x) \circ_2 h_i(y)$, with $1 \leq i \leq n$ and n is the number of components. Then, NotInjNotIso has to show for each equation system separately that $h_i(c_1) \doteq h_i(c_2)$ 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.

9.2.4 Automatically Classifying Isomorphic Structures

Similar to the exploration module for simple properties of residue class structures (see section 9.1.5) we implemented an exploration module in ΩMEGA that divides a given set of residue class structures into disjunct classes of isomorphic structures. The module takes the first given structure and creates an isomorphism class that contains only this structure. Then, it starts to perform 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: The exploration module first performs a computation whose result gives 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. Thereby the exploration module employs the same functionality as the control rule ComputeInstbyCasAndMG when it constructs a pointwise defined function (see section 9.2.1).

As opposed to the classification described in section 9.1.5, the exploration module does not construct and discharge a proof obligation of each check. Instead, it first conducts all possible checks and then construct proof obligations. If the exploration module finds an S' to which S is supposedly isomorphic, then it constructs this proof obligation. Otherwise, it constructs for each isomorphism class C a proof obligation that S is not isomorphic to a $S' \in C$. This way the exploration module postpones and even avoids 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 or SEM's result the algorithm proceeds by constructing the negated proof obligation and passes it again to MULTI to discharge it. In case this proving attempt fails, too, the algorithm signals an error.

9.3 Results and Discussion

We conclude this chapter with a discussion of the conducted case study and its results. The section is structured as follows. First, we discuss related work. Afterwards, we give in section 9.3.2 an account of the experiments conducted in the residue class domain. In section 9.3.3, we evaluate the realized proof planning approach. Finally, we compare our multiple strategy proof planning approach in the residue class domain with the application of an automated theorem prover to the same problems in section 9.3.4.

9.3.1 Related Work

Combining Computer Algebra and Theorem Proving

There are various accounts on experiments of combining computer algebra and theorem proving in the literature, see [131] for just a few. We can distinguish between two major paradigms for these integrations: (1) The integration of deduction into computer algebra and, conversely, (2) the use of computer algebra during theorem proving. Most of this existing work deals with the technical and architectural aspects of those integrations as well as with correctness issues.

In this case study we use two computer algebra systems in proof planning. Previous work in this area is reported in [135] and [222]. Both papers present the integration of computations of computer algebra systems within methods (e.g., ComplexEstimate-B in [222]) and explain how the correctness of certain limited computations of a computer algebra system such as Maple can be guaranteed within the proof planning framework. We did make use of this previous work when implementing methods such as SolveEquation-b, which calls Maple to check equations. But in this case study we mainly focus on the integration of computer algebra systems to provide instantiations for meta-variables.

Theorem Proving in Abstract Algebra

For the particular domain of abstract algebra [124] sketches a possible cooperation between the deduction system Nuprl and the computer algebra system Weyl. Other work in theorem proving in this domain concentrates mainly on the equational reasoning aspect in abstract algebra. As examples we refer to term rewrite systems for finite groups as presented for instance in [36] and to the specialized superposition calculi for groups in [226] and for monoids in [92].

Exploration in Finite Algebra

Work on exploration and automated discovery in finite algebra is reported in [90, 150, 219, 252] where model generation techniques are used to tackle quasi-group existence problems. In particular, systems such as Finder [218] and Sato [251] were successfully employed to solve some open problems in quasi-group theory. [153] gives an account of the use of the automated theorem prover Otter to assist the construction of non-associative algebras in every day mathematical practice. Other work [103] employs constraint solving techniques to complete quasi-group multiplication tables. The motivation for all this work is roughly to specify certain properties of an algebra and then to try to automatically construct a structure that satisfies the required properties. Thus, the constructed algebra might actually be a new discovery. Our work is diametrical in the sense that we start out with given structures and classify them with respect to their algebraic properties and whether they are isomorphic.

Constructing Discriminants with HR

There are several other applications to perform categorization tasks with HR. In [60] a heuristic search is performed within HR, which measures the concepts in various ways and builds new concepts from the most interesting old ones first. [61] discusses the usage of a forward look ahead mechanism, which can tell in advance whether the application of up to three concept formation steps will lead to a concept which achieves a particular categorization task (e.g., a discriminant).

The problem of identifying a discriminant for two objects is a machine learning problem and could, in theory, be solved by a program such as Progol [177]. Progol uses Inductive Logic Programming to identify a concept which correctly categorizes

	Simple Properties				Iso-Classes		
	All	$\mathbb{Z}_5$	$\mathbb{Z}_6$	$\mathbb{Z}_{10}$	$\mathbb{Z}_5$	$\mathbb{Z}_6$	$\mathbb{Z}_{10}$
Magmas	8567	3049	4152	743	36	7	14
Abelian Magmas	244	53	73	24	26	5	6
Semi-groups	2102	161	1114	35	3	8	1
Abelian Semi-groups	2100	592	1025	62	1	12	2
Quasi-groups	1891	971	738	70	9	2	10
Abelian Quasi-groups	536	207	257	11	3	2	1
Abelian Monoids	211	97	50	6	1	1	1
Abelian Groups	1001	276	419	49	1	1	1
Total	18963	5406	8128	1000	80	38	36

Table 9.3: Results of the experiments.

a set of positive and negative examples. However, as mentioned in [59], this may be difficult in practice in our setting since we supply only a single positive and a single negative example, which would suggest that the amount of compression in a concept would not be high enough to be suggested as a viable solution.

Randomization and Restart Techniques

Recent work in Artificial Intelligence demonstrates that several hard combinatorial search procedures show heavy-tailed behavior and that randomization and restart techniques can help to boost the search as well as to solve formerly unsolved problem classes. [105] describes the application of the technique on scheduling problems in a constraint satisfaction formulation (CSP); [104] demonstrates the effectiveness of the technique on propositional satisfiability (SAT) and CSP algorithms in the domains of logistics planning, circuit synthesis, and round-robin scheduling; finally, [103] describes additional results in the domain of the so-called quasi-group completion problem (in a CSP formulation), school time tabling (in a SAT formulation), and problems from the Dimacs Challenge benchmark (in a SAT formulation). As opposed to these heavy-tailed search problems, the blocks-world planning domain does not show heavy-tailed behavior (see [104]).

To the best of our knowledge, randomization and restart techniques were employed in deduction systems only in propositional SAT provers (see [104]). Ertel describes in [80] the competitive application of randomized strategies of the SETHEO theorem prover (see also section 6.4.2). However, this approach is not based on the analysis of underlying cost distributions.

9.3.2 Tests

To test the realized strategies we constructed a large testbed of automatically generated problems about residue classes modulo n, where n ranges from 2 to 10, together with operations that are systematically constructed from the basic operations $\bar{+}, \bar{-}, \bar{*}$. Altogether, we have classified 18.963 structures with respect to their algebraic properties so far, including a large set of structures concerning the sets $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$. The results for all explorations as well as for each of $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$ are given on the left hand side of Table 9.3. The table shows the number of structures we have found in each algebraic category; the table omits those algebraic categories for which we have not found any representative (i.e., loops, non-Abelian monoids and groups). Note that the total number of explored structures also includes some that were not closed, which are not displayed as a separate category.

To show the validity of the techniques for isomorphism and non-isomorphism proofs we applied our classification process to the structures involving $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$. We only classified those structures belonging to the same algebraic category; that is, *a priori* we excluded the comparison of magmas and semi-groups etc. The different isomorphism classes we have found so far for the structures of each category are given on the right hand side of Table 9.3.

In the experiments, we were interested to prefer the application of the strategies ReduceToSpecial and EquSolve before TryAndError since they produce shorter and more elaborate proofs. For the simple properties, Multi could successfully employ ReduceToSpecial to a sample of 20%, EquSolve for 23% of the proofs, and the remaining 57% of the examples could only be solved by the TryAndError strategy. These figures are not as disappointing as they seem at first glance if we consider 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. From the necessary isomorphism proofs 88% were constructed with the EquSolve strategy, the other 12% were constructed with TryAndError. During the automatic classification 1276 non-isomorphism proofs were constructed. Here 18% of the proofs were done by finding a discriminant[13]; the remaining 82% with the NotInjNotIso strategy.

Although from a theoretical point of view all proof plans can be constructed by exhaustive search without employing strategies of **InstMeta**, in practice the combinatorial explosion makes this infeasible. Thus, reliable and robust instantiation strategies are crucial for the success of Multi in this domain. Indeed, we have not found a single case where the instantiations provided by GAP, Maple, or SEM have failed or were incorrect for the proofs of simple properties. The situation is somewhat different for the isomorphism problems. The classification process as well as the instantiation of meta-variables in the strategy ComputeInstbyCasAndMG depend on the quality of Maple's and SEM's solutions for the system of instantiated homomorphism equations. It turned out that Maple sometimes does not return all possible solutions even though it was asked to do so. For instance, the two structures $(\mathbb{Z}_6, \lambda xy_\blacksquare \bar{2}_6 \bar{*} x \bar{*} y)$ and $(\mathbb{Z}_6, \lambda xy_\blacksquare \bar{4}_6 \bar{*} x \bar{*} y)$ are isomorphic (a possible isomorphism is $h(x) \doteq \bar{5}_6 \bar{*} x$). When called to give the solutions for the corresponding set of instantiated homomorphism equations, Maple returns the mapping $h(x) \doteq \bar{0}_6$ as sole solution. Although this is a correct solution, it is not the only one. In particular, it is not suitable to construct an isomorphism necessary for testing in the classification process and for providing a pointwise function as instantiation of meta-variables. Actually, during our experiments, Maple failed to compute all solutions and hence to give suitable pointwise functions for about 2% of the queries. Unfortunately, we could not find a clear characterization of these cases in order to work around the problem. SEM never failed to provide suitable and correct pointwise functions during our experiments. The drawback of SEM is that it cannot produce closed polynomial representations of isomorphisms as needed to apply the EquSolve strategy. Maple and SEM can cooperate by passing the pointwise isomorphisms provided by SEM to Maple to create a corresponding polynomial representation.

[13] The technique for finding a discriminant with HR described in section 9.2.2 was implemented after these experiments were already finished. In the setting of the experiments we used only two pre-defined discriminants which were contained in theorems that are applied by ReduceToSpecial (see [162] for a detailed description of this technique). We assume that with the fully implemented discriminants technique a considerably larger part of the non-isomorphism problems can be solved by this technique.

9.3.3 Evaluation of the Proof Planning Approach

To avoid that the proof planning approach is too fine tuned to initial examples (see BUNDY's critique quoted in section 8.4.3) we developed the proof planning approach to tackle residue class problems on the basis of a relatively small number of examples. Afterwards, we tested the realized approach against a large number of examples that differ from the initial examples used during the design process.

In detail, we used 21 examples to design the basic versions for the simple property problems of the ReduceToSpecial, TryAndError, EquSolve, and ComputeInstbyCasAndMG strategies. For the extensions to handle direct products we used 3 additional examples; for the extensions to classify structures with two operations we needed 2 examples, which were combinations of already used examples. We used 15 examples to develop the additions to the ReduceToSpecial, TryAndError, EquSolve, and ComputeInstbyCasAndMG to handle isomorphism and non-isomorphism problems and another 4 examples to build the NotInjNotIso strategy.

Our tests (see section 9.3.2) provide evidence that

- our techniques realized in the strategies provide a robust machinery suitable to prove a large variety of problems about residue classes,
- the integration of computer algebra, model checking, and theory formation systems enhances indeed the proof planning process,
- elaborate techniques such as the construction and proof of discriminants result in proof objects that are very similar to human proofs for residue class problems.

In the following, we shall discuss the strategies, methods, and control rules developed for the residue class domain with respect to their amount of mathematical and domain-specific content. Moreover, we shall discuss the generality of the single strategies, methods, and control rules, i.e., to which domains they can be applied, as well as the generality of the encoded principles.

TryAndError

The TryAndError strategy fits into the more general heuristic strategy "split into an exhaustive set of cases, then solve single cases".[14] It instantiates this mathematical principle with the specific knowledge on how to apply it to residue class problems. This principle is suitable for our domain since the quantified variables range only over finite domains. The same technique may be used to tackle other domains of finite group theory or finite algebra. The second basic principle of TryAndError is to solve the single cases by reducing statements on residue classes into statements on integers and to solve the statements on integers by numerical reasoning. This is a domain-specific principle that resembles human approaches to solve residue class problems.

The method $\vee$E**-B, which performs a case-split with respect to a set of disjunctive supports, is a general, logic-level method without particular mathematical content. The mathematical knowledge of how to organize the exhaustive case analysis is encoded in the control rule `tryanderror-standard-select` (see Figure 4.4 in section 4.1.3) that guides the application of $\vee$E**-B and some domain-specific methods for residue class theorems. Control rules guiding exhaustive case analysis

[14] SCHOENFELD mentions case analysis as a frequently used heuristic: "Decompose the domain of the problem and work on it case by case." ([209] p. 109)

in other domains could be similar to `tryanderror-standard-select`. That is, they could use also $\vee$E**-B but combine it with different domain-specific methods.

The methods $\forall$IRescalss-B, $\exists$IResclass-B, and ConCongCl-B encode the mathematical knowledge on how to reduce statements on residue classes to statements on integers; ConCongCl-B reduces equations and other quantifier-free statements whereas $\forall$IRescalss-B and $\exists$IResclass-B reduce quantified statements. All three methods are domain-specific for residue class problems and can hardly be used to tackle other problem classes.

$\forall$IRescalss-B and $\exists$IResclass-B combine the decomposition of the quantifier with a representation-shift. We illustrate this with the example depicted in Figure 9.1 in section 9.1.1. A domain-independent method for the decomposition of a universal quantifier would reduce the goal $\forall x{:}\mathbb{Z}_2.\exists y{:}\mathbb{Z}_2.(x\bar{+}y\doteq\bar{0}_2)\wedge(y\bar{+}x\doteq\bar{0}_2)$ in L_{24} to $\exists y{:}\mathbb{Z}_2.(c'\bar{+}y\doteq\bar{0}_2)\wedge(y\bar{+}c'\doteq\bar{0}_2)$ with a new hypothesis $c'\in\mathbb{Z}_2$. As opposed thereto, $\forall$IRescalss-B represents the c' of the general method as $cl_2(c)$ in both, the new goal and the new hypothesis (see L_{23} and L_1 in Figure 9.1). As result, $\forall$IRescalss-B and $\exists$IResclass-B are over-specific in the sense that their functionalities could be realized by the combination of two more general methods, i.e., a general method for quantifier decomposition and a method for representation-shifts. We decided for the integrated representation-shift in $\forall$IRescalss-B and $\exists$IResclass-B since the separated representation-shift turned out to be tedious and results in unintuitive proof plans.[15] There is an ongoing PhD by Martin Pollet that addresses (among others) the question of the incorporation and use of different representations of mathematical objects in proof planning. Hopefully, operations like representation-shifts will become better supported by the techniques developed in this PhD.

Similarly, also the methods $\forall$IResclFunc-B and $\exists$IResclFunc-B for decomposing quantifier that range over functions of residue class sets are over-specific. They also combine the domain-independent decomposition of the quantifier with domain-specific representation-shifts.

As result, the decomposition of quantifiers and connectives in TryAndError is domain-specific and part of the domain knowledge (in particular, the decomposition of disjunctive supports by $\vee$E**-B). Therefore, TryAndError (as well as EquSolve and NotInjNotIso) does not employ the general strategies UnwrapHyp and NormalizeLineTask known from the limit domain for the decomposition of quantifiers and connectives, but rather employs domain-specific methods and a domain-specific control.

Altogether, TryAndError is not restricted to the classification problems discussed in this chapter. Its principle "split into an exhaustive set of cases, then solve single cases" can tackle any statements on residue classes whose quantifiers range over finite residue class sets. For instance, it can prove the discriminant properties introduced by HR.

EquSolve

Similar to TryAndError, EquSolve relies on the principle "reduce statements on residue classes to statements on integers". It combines this domain-specific principle with the more general principle "solve the resulting statements on integers by

[15] Technically, the representation-shift from c' to $cl_2(c)$ uses the theorem $\forall x{:}\mathbb{Z}_n.\exists y{:}\{0,\ldots,n-1\}.x = cl_n(y)$ from the residue class domain. Each application of this theorem for the same $c'\in\mathbb{Z}_n$ introduces a new constant for the y. Because of our ND-calculus biased framework we would have to apply the theorem to each proof branch separately. This would result in several $cl_2(c_1)$, $cl_2(c_2)$, ... representations for the same initial $c'\in\mathbb{Z}_2$. To complete the representation-shift TryAndError would have to prove that all resulting $c_1, c_2, \ldots$ are equal and would have to replace all occurrences of $c_1, c_2, \ldots$ by one constant.

equational reasoning". This second principle is applicable also to other domains that rely on equations.

The combination of the two principles was successful for the residue class domain since we could employ the computer algebra system MAPLE to solve equations on integers. We encoded the knowledge on how to exploit (the knowledge in) MAPLE into the method SOLVEEQUATION-B. SOLVEEQUATION-B is not restricted to the residue class domain but can be employed in any domain with equations on integers.

Also the method ISOTODISCRIMINANT-B in EquSolve is not restricted to the residue class domain. Rather, it covers the general mathematical knowledge on how to accomplish non-isomorphism proofs with discriminants.

Altogether, EquSolve is not as general as TryAndError since it can handle only such problems of the residue class domain that can be reduced to equations. However, similar to TryAndError, it is not restricted to the classification problems discussed in this chapter. For instance, it can also solve subproblems on discriminant properties resulting from the application of HR.

NotInjNotIso

NotInjNotIso is specialized to one type of problems of the residue class domain, namely non-isomorphism problems. Its basic principle "assume negation of theorem, then create contradiction" of constructing indirect proofs is a general proof paradigm known from mathematics.

NotInjNotIso implements this general principle by equational reasoning with the set of instantiated homomorphism equations in order to derive the contradiction. This equational reasoning by applying instantiated homomorphism equations with the general, logic-level method =Subst-B could also be used to tackle non-isomorphism problems in other domains. The selection of the next equation to apply in the control rule `choose-next-equation` and the guidance of the cutoffs and restarts in the control rules `interrupt-if-cutoff` and `restart-NotInjNotIso` are domain-specific. Whereas `choose-next-equation` exploits the mathematical knowledge of which equations support canceling (see section 9.2.2), `interrupt-if-cutoff` and `restart-NotInjNotIso` encode stochastic knowledge, which we acquired by extensive experiments, of when NotInjNotIso should be interrupted and restarted.

The cutoff and restart knowledge itself (i.e., the concrete values for cutoffs and restarts) cannot be directly transfered to other domains. However, the approach we used to acquire this knowledge is domain-independent and was applied already to several hard Artificial Intelligence search problems (see discussion of related work in section 9.3.1).

ReduceToSpecial

We used the domain-independent strategy ReduceToSpecial already to tackle limit problems. There it turned out that some domain-specific control was needed to guide the applications of some theorems of the limit domain (see section 8.3).

When we applied ReduceToSpecial to the residue class domain, we found that the general theorem application method APPLYASS-B was not sufficient to apply all theorems of the residue class domain. To overcome these problems we implemented further methods to decide the applicability of different theorem classes (see section 9.1.3). These new methods contain no particular mathematical or domain-specific knowledge but rather employ different specialized algorithms deciding particular higher-order unification problems. It is not yet clear how general these methods and algorithms are, i.e., whether they can be used to tackle other domains. However, it is clear that specialized algorithms deciding particular higher-order unification problems will be helpful in other domains as well.

ComputeInstbyCasAndMG **and** ComputeInstbyHR

The **InstMeta** strategies ComputeInstbyCasAndMG and ComputeInstbyHR interface computer algebra systems, a model checker, and a theory formation system. These strategies contain the knowledge of how to exploit the specific knowledge in the connected external systems in order to compute instantiations for meta-variables.

The implemented functionalities of ComputeInstbyCasAndMG are currently focused on the residue class case study (i.e., what kinds of meta-variables are recognized and what kind of computations are requested from the connected systems). However, the principle of ComputeInstbyCasAndMG to search for facts in the proof plan that determine the needed kind of instantiation for a meta-variable and to employ then suitable experts to compute a concrete instantiation is a general principle that can be easily extended to tackle also other domains and other problems. For instance, when another kind of meta-variable instantiation is needed, then further computations using the current external systems could be added. Moreover, ComputeInstbyCasAndMG could interface further external systems.

As opposed thereto, the functionality of ComputeInstbyHR is currently very restricted. It recognizes only one kind of problems. We could have implemented the functionality of ComputeInstbyHR as a part of ComputeInstbyCasAndMG (then ComputeInstbyCasAndMG would have to interface HR). We decided, however, to further examine the integration of theory formation systems such as HR into proof planning with further kinds of examples before we determine the principle of how they are connected.

9.3.4 Comparison with ATPs

The successful application of proof planning to problems of a mathematical domain depends on the acquisition of mathematical knowledge of the domain and its formalization in methods, control rules, and strategies. If suitable knowledge is available, proof planning can solve problems that are beyond the means of traditional ATPs based on general-purpose machine-oriented logical calculi such as the resolution calculus [205]. If the number of problems of a domain is sufficiently large, the acquisition of the knowledge and its formalization can prove fruitful but is nevertheless a tedious task.

This poses the question of whether there are other means than proof planning to tackle the problems of a certain domain. The problems generated during the exploration of residue class structures are in the range of traditional automated theorem proving since all occurring quantifiers range over finite sets. To compare the results of our combined proof planning, Maple, GAP, HR approach with the results of a traditional automated theorem prover we applied the first order equational prover WaldMeister [114] to the same problems. In order to guarantee a fair comparison we were interested to exploit expert knowledge about suitable control settings for automated theorem provers and suitable formalizations of the problems.[16] We decided for WaldMeister since we got help from one of its implementors in tuning the system for our problems.

[16] Indeed, some experiments showed that, without expert knowledge about suitable control settings for the systems and suitable formalizations of the problems, we were hardly able to solve any of our problems.

9.3.4.1 Proving Residue Class Problems with WALDMEISTER

We employ WALDMEISTER in an **ATP** strategy, WaldOnResidueClass, which applies WALDMEISTER to a line-task. The strategy can be applied to all problems occurring during the automatic exploration except to show that two structures are isomorphic. The application function of WaldOnResidueClass creates input files for WALDMEISTER that consist of three parts: A general axiomatization of the residue class structure and the operations $+, -, *$, a specific formalization of the property to be proved, and a suitable control setting for WALDMEISTER, for instance, an order of symbols. The strategy WaldOnResidueClass calls WALDMEISTER with two different control settings depending on whether the goal to be proved is a simple property or a non-isomorphism problem. The output of WALDMEISTER when employed by WaldOnResidueClass cannot be translated into an ND-proof by TRAMP since the input for WALDMEISTER (and hence also its output) comprises facts for which we have no corresponding facts in ΩMEGA's database. Thus, the output check function of WaldOnResidueClass just checks whether WALDMEISTER declares in its output the problem as proved.

$$
\begin{array}{ll}
\left.\begin{array}{l}
a_0 = 0\\
a_1 = s(a_0)\\
equal(x,x) = true\\
equal(x,s(x)) = false\\
s(s(x))) = x\\
Z_2 = cons(a0, cons(a1, nil))
\end{array}\right\}(1) & \begin{array}{l}\text{Specification of } \mathbb{Z}_2 \text{ as}\\ \text{list of two elements.}\end{array}\\
\left.\begin{array}{l}
+(x,0) = x\\
+(x,s(y)) = s(+(x,y))\\
\vdots\\
*(x,0) = 0\\
(x,s(y)) = +(x,(x,y))\\
\vdots
\end{array}\right\}(2) & \begin{array}{l}\text{Specification of the}\\ \text{basic operations } +,*,-.\end{array}\\
\left.\begin{array}{l}
+(+(x,y),z) = +(x,+(y,z))\\
+(x,y) = +(y,x)\\
\vdots\\
-(-(x)) = x\\
-(x,-(x)) = 0\\
\vdots\\
((x,y),z) = *(x,*(y,z))\\
(x,+(y,z)) = +((x,y),*(x,z))\\
\vdots
\end{array}\right\}(3) & \begin{array}{l}\text{Additional theorems}\\ \text{and lemmas about}\\ \text{the basic operations.}\end{array}\\
\left.\begin{array}{l}
op(x,y) = *(+(x,s(0)),y)
\end{array}\right\}(4) & \begin{array}{l}\text{Specification of the operation}\\ \text{of the residue class structure.}\end{array}
\end{array}
$$

Figure 9.14: Specification for WALDMEISTER.

Figure 9.14 depicts the general part of the input specification for the example $(\mathbb{Z}_2, \lambda xy_{\bullet}(x \bar{+} \bar{1}_2) \bar{*} y)$. The general part consists of facts that (1) model the residue class set $\mathbb{Z}_2$ as a list of elements, (2) model the basic operations $+, -, *$, and (3) add useful known lemmas and theorems about the basic operations such as the

	$\mathbb{Z}_5$	$\mathbb{Z}_{10}$
Explorations wrt. to simple prop.	1100	316
Failed Explorations	49	247
Single simple property problems	4694	1260
Failed simple properties problems	53	314
Non-isomorphism problems	2400	400
Failed non-isomorphism problems	167	65

Table 9.4: Results of applying WALDMEISTER to problems of $\mathbb{Z}_5$ and $\mathbb{Z}_{10}$.

ring properties.[17] The operation $\lambda xy.((x\bar{+}\bar{1}_2)\bar{*}y)$ can then be expressed directly by these functions (part (4) in Figure 9.14). In this specification, the multiplication table of the structure does not need to be formalized. We experimented also with an explicit specification of the multiplication table of the structures, similar to the problem specifications for SEM. However, WALDMEISTER performed better when the operation of the residue class structure was defined as a composition of basic operations. The reason is that the knowledge of the basic operations given as lemmas in part (3) of the specification are crucial for success. If the operation is specified via its multiplication table, then it is not possible to provide WALDMEISTER with lemmas on the operation.

To prove simple properties, we have to define the property in question recursively over the list specifying the structure of the actual problem. This can only be done by introducing several auxiliary predicates. The theorem to be proved by WALDMEISTER is an equation stating that the simple property does or does not hold.

To show that two structures are not isomorphic WaldOnResidueClass uses WALDMEISTER to construct an indirect proof. That is, to the specification of the two structures $(RS^1_n, \circ^1)$ and $(RS^2_m, \circ^2)$, the definition of two homomorphisms

$$h : RS^1_n \to RS^2_m \text{ and } j : RS^2_m \to RS^1_n$$

and the properties $h(j(x)) = x$ and $j(h(x)) = x$ are added. The theorem to be proved by WALDMEISTER consists of all possible equations between two distinct elements of RS^2_m, such as 0=s(0), etc. If WALDMEISTER succeeds to prove that one of these equations holds, then we have a contradiction to the assumption that the two structures are isomorphic.

9.3.4.2 Experiments

To compare the proof planning approach (combined with MAPLE, GAP, SEM, HR) with the application of WALDMEISTER we used WALDMEISTER to explore structures with the sets $\mathbb{Z}_5$ and $\mathbb{Z}_{10}$, which we already classified with respect to their simple algebraic properties in our experiments reported in section 9.3.2. Moreover, we tackled non-isomorphism problems with the sets $\mathbb{Z}_5$ and $\mathbb{Z}_{10}$. The results of our experiments are summarized in Table 9.4. All experiments were conducted on a Sun Sparc Ultra with four processors and 2 GB Ram; the maximum time bound for WALDMEISTER was 1500 seconds.

Our experiments show that WALDMEISTER is generally able to solve all considered problems in the residue class domain. However, it turned out that on a large testbed WALDMEISTER is less robust than our proof planning approach. WALDMEISTER failed on 4% of the $\mathbb{Z}_5$ and 78% of the $\mathbb{Z}_{10}$ explorations. The most brittle categories are the non-associative problems for $\mathbb{Z}_5$, for which WALDMEISTER failed

[17] In the specifications for WALDMEISTER '$-$' is a unary function. Thus, our binary minus operation is translated as $+(x, -(y))$.

on 49 of 888 problems, and divisors and non-divisors problems for $\mathbb{Z}_{10}$, for which WaldMeister failed on 39 of 39 problems and 197 of 223 problems. Note that this does not necessarily mean that WaldMeister might not be able to prove these problems at all if it were given a more specialized and fine tuned control setting. In our experiments, however, we use only two control settings, one suitable for all simple properties and one for non-isomorphism problems. According to our experiments, the overall performance of WaldMeister (i.e., whether it succeeds or fails on a problem) depends on the cardinality of the set involved: higher cardinality implies a higher likelihood of failure.

9.3.4.3 Discussion

WaldMeister has a clear advantage over the proof planning approach with respect to runtime behavior. When it succeeds, it succeeds very fast independently of the cardinality of the residue class structure (30% of all proofs were produced in less than 1 second, 70% of all proofs were produced in less than 10). The runtime performance of proof planning depends on which strategy can be applied successfully. Problems solved with the ReduceToSpecial or the EquSolve strategy usually take about 10 to 20 seconds independently of the cardinality of the given set. If TryAndError has to be applied, it can take considerably longer, depending on the cardinality of the structures.

In our context, a disadvantage of WaldMeister is its output format. Although WaldMeister has a proof presentation tool that tries to structure the found proof by lemmas, in our experiments this tool failed to successfully present many found proofs (e.g., on almost all associativity problems). And even proofs displayed by the presentation tool are relatively hard to read: on the one hand, the proofs are very long, usually between 150 and 300 equational reasoning steps, structured with 10 to 30 lemmas. On the other hand, the lemmas are rather counterintuitive for humans. In contrast, the proof planning approach can produce very short $\mathcal{PDS}$s when ReduceToSpecial ($\sim$ 10 proof lines) or EquSolve ($\sim$ 20 proof lines) are applied. Although proof plans with TryAndError can be very long, these proofs are structured in a clear way by the case-splits. For instance, a divisors proof for a structure with cardinality 10 consist of about 3000 nodes comprised of 100 clearly separate cases each consisting of about 30 steps.

It is a common criticism on proof planning (e.g., see [42]) that it depends on specially prepared domain knowledge. This criticism assumes that automated theorem provers such as WaldMeister do not depend on particular knowledge since they are based on general-purpose machine-oriented calculi. However, our experience with WaldMeister is that its application to our domain was successful only with a considerable amount of very specific knowledge. The WaldMeister strategy WaldOnResidueClass comprises, for instance, the technical knowledge of how to suitably represent residue class structures for WaldMeister, knowledge of which lemmas for the basic operations to add, and knowledge of which particular order of the symbols to choose. This knowledge is absolutely crucial for a successful application of WaldMeister in our domain. Instead of encoding mathematical knowledge for the residue class domain, we had to encode knowledge specific to the theorem prover employed, which we could only do with the help of an expert.[18] We failed to successfully apply the first-order resolution prover Otter [150] in our domain since we lacked the expert knowledge to find a suitable representation for our problems.

[18] In the field of term rewriting systems there is knowledge of orders and representations for fragments of Peano Arithmetic (e.g., see [11, 10]) that provides a starting point for developing control settings for new applications. The selection of lemmas requires experience with the concrete system and its underlying algorithm.

Chapter 10

Further Applications of MULTI

In this chapter, we shall briefly discuss two further case studies conducted with MULTI. In the first case study we apply MULTI to solve problems of permutation groups. Here MULTI performs hierarchical proof planning with unreliable methods whose applications have to be expanded with the expansion strategy ExpS. In the second case study we tackle homomorphism theorems with MULTI. Although these theorems can be solved automatically with MULTI, the focus in this case study is to use MULTI for interactive theorem proving.[1]

We shall briefly discuss these two case studies in the following two sections, respectively, since they address expansion and interactive theorem proving with MULTI, two issues that are not addressed by the two large case studies described so far.

10.1 Proof Planning Permutation Group Problems

The permutation group domain consists of different kinds of problems concerned with properties of permutations and permutation groups. Essential for the success of MULTI in this domain is the incorporation of the computer algebra system GAP. As in the residue class domain, GAP can provide suitable instantiations of occurring meta-variables that simplify the problems at hand considerably. The main strategy to tackle permutation group problems is the **PPLANNER** strategy PermStrat. The cooperation of PermStrat with GAP works analog to the incorporation of computer algebra systems in the residue class domain: for occurring meta-variables PermStrat interrupts and places demands for the **INSTMETA** strategy InstPermTH-FromGap, which queries GAP to provide suitable instantiations.[2]

We start with a brief introduction into computational permutation group theory and its formalization in ΩMEGA. Afterwards, we illustrate with an example how

[1] The case study on permutation groups was conducted by Martin Pollet and Volker Sorge from the ΩMEGA group together with Arjeh Cohen and Scott Murray from the Technische Universiteit Eindhoven, Netherlands. The contribution of the author of the thesis to this case study consisted only of providing functionalities in MULTI and technical support for the application of MULTI.

[2] Technically, InstPermTHFromGap considerably differs from ComputeInstbyCasAndMG. The reason is that, as opposed to the residue class domain where we use only functionalities directly offered by MAPLE and GAP, InstPermTHFromGap has to provide GAP with new functions for the permutation group domain. Only with these new functions GAP can provide certificates for queries from which InstPermTHFromGap can then compute the needed instantiations.

MULTI performs hierarchical proof planning in this domain. Thereby, we focus on the expansion issue. A more detailed description of the permutation group domain and how it is tackled with MULTI and GAP can be found in [57].

Computational Permutation Group Theory

In computational permutation group theory, a group G is specified by a list of generating permutations $A = \{a_1, \dots, a_k\}$ where a_i is a permutation on the points $\Omega := \{1, 2, \dots n\}$. We also write $G = \langle A \rangle$ to denote that G is generated by A. While there are different notations in mathematics to express permutations, the cycle notation is usually preferred. In this notation a permutation consist of duplicate-free disjoint cycles, that is, lists $(n_1, \dots, n_k)$ of points with $n_i \neq n_j$ for $i \neq j$. A cycle maps the point n_i to n_{i+1} for $i = 1, \dots, k-1$ and n_k to n_1. A permutation is then either a set containing disjoint cycles or the composition of permutations. For instance, the so-called *Mathieu group* on 11 points, denoted by M, is generated by the list $A = \{a_1, a_2\}$, where: $a_1 = (1,10)(2,8)(3,11)(5,7)$, $a_2 = (1,4,7,6)(2,11,10,9)$.

A permutation g belongs to the group $G = \langle A \rangle$ where $A = \{a_1, \dots, a_k\}$, if there is a word of the form $g = a_{i_1}^{e_1} a_{i_2}^{e_2} \cdots a_{i_m}^{e_m}$ where the indices i_j are in the range $1, \dots, k$ and the exponents e_j are integers. For instance, for the group M and $g = (1,3,8,9)(4,10,6,5)$ the word that certificates that $g \in M$ is $a_1 {a_2}^3 a_1$.

Formalization and Problems

Objects in the permutation group domain are formalized as follows. A cycle has the basic type *cyc*. A permutation is a set of cycles and has thus the type $cyc \to o$.[3] A permutation group G that is constructed by a set of generating permutations has type $(cyc \to o) \to o$. The generator $\langle\ \rangle$ has type $((cyc \to o) \to o) \to (cyc \to o) \to o$. The operation of a permutation group, $\circ$, is the composition of permutations. $\circ$ has the type $(cyc \to o) \to (cyc \to o) \to cyc \to o$. We have a special operator for the application of a permutation to an element of the underlying set Ω, namely #. Since Ω is a set of elements of type ν, # has the type $(cyc \to o) \to \nu \to \nu$.

The permutation group domain consists of different kinds of problems (see [57] for a complete description of the domain) among them are:

Membership Given a permutation g and a permutation group $G = \langle A \rangle$, show that $g \in G$.

Orbit-Exists Given a permutation group $G = \langle A \rangle$ and a point $x \in \Omega$, determine the orbit of G with respect to x (i.e., find $Gx \subset \Omega$ with $Gx = \{g\#x : g \in G\}$).

Orbit-Membership Given an orbit Gx and $y \in \Omega$, show that $y \in Gx$.

Points-Closed Given a permutation g and a subset S of the point set Ω, show that S is closed with respect to g, that is, show that for all $y \in S$ $g\#y \in S$.

The concept *Orbit* is formalized in ΩMEGA's database with two type variables α and β:

$$Orbit_{(\alpha o)(\alpha\beta\beta)\beta\beta o} \quad \equiv \quad \lambda G_{\alpha o} \centerdot \lambda f_{\alpha\beta\beta} \centerdot \lambda x_\beta \centerdot \lambda y_\beta \centerdot \exists g_\alpha{:}G \centerdot y \doteq f(g,x)$$

Here α is the type of the elements of G and β is the type of the points in Ω. In the permutation group domain α is $cyc \to o$ and β is ν. Thus, the term $Orbit(\langle\{a_1,a_2\}\rangle_{(cyc\to o)\to o}, \#_{(cyc\to o)\to\nu\to\nu}, 1_\nu)$ has the type νo.

[3] To avoid confusion we write composed types containing *cyc* with arrows, e.g., $cyc \to o$ instead of *cyco*.

We distinguish in the permutation group domain simple problems and complex problems. Simple problems are such problems that occur as subproblems of other problems. For instance, in the example we shall discuss below membership, orbit-membership, and points-closed problems are simple subproblems whereas the main problem is an orbit-exists problem. We use hierarchical proof planning in the permutation group domain to hide proofs of the simple problems when they occur as subproblems of complex problems. This allows to come up fast with abstract proof plans for complex problems. The tedious details whose construction can nevertheless be very time consuming are delayed until the expansion.

The PermStrat Strategy

Technically, this is realized by unreliable methods in the strategy PermStrat that close a simple problem immediately. For instance, PermStrat contains the methods PermInGroup-B, OrbitMember-B, and PointsClosed-B, which close proof lines that state membership, orbit-membership, or points-closed problems. A strategic control rule delays the expansion-tasks arising from the application of an unreliable method until all line-tasks are closed. Then, Multi applies the **Exp** strategy ExpS to expand these steps. The expansion re-opens the simple subproblems and Multi applies again PermStrat to them. PermStrat contains a control rule that forbids to apply a method to a goal if there is already an justification of this method for the goal at a higher level of abstraction (i.e., if the goal was already justified by an application of this method and this justification was already expanded). This control rule forbids the application of the same unreliable methods to the re-opened subproblems, and PermStrat has to construct a proof plan with other methods for the re-opened subproblems.

L_{22}.	$\vdash \forall y{:}\{1,\ldots,11\}\blacksquare(a_2\#y) \in \{1,\ldots,11\}$	(PointsClosed-B)
L_{21}.	$\vdash \forall y{:}\{1,\ldots,11\}\blacksquare(a_1\#y) \in \{1,\ldots,11\}$	(PointsClosed-B)
L_{20}.	$\vdash \forall y{:}\{1,\ldots,11\}\blacksquare(a_2\#y) \in \{1,\ldots,11\} \wedge$	($\wedge$I-B L_{21} L_{22})
	$\forall y{:}\{1,\ldots,11\}\blacksquare(a_1\#y) \in \{1,\ldots,11\}$	
L_{18}.	$\vdash 1 \in \{1,\ldots,11\}$	(InSet-B)
L_{19}.	$\vdash \forall z{:}\{a_1,a_2\}\blacksquare\forall y{:}\{1,\ldots,11\}\blacksquare(z\#y) \in \{1,\ldots,11\}$	($\forall$I-FiniteSort-B L_{20})
L_{17}.	$\vdash \forall z{:}Orbit(\langle\{a_1,a_2\}\rangle,\#,1)\blacksquare x \in \{1,\ldots,11\}$	(FixPoint-B L_{18} L_{19})
L_3.	$\vdash Orbit(\langle\{a_1,a_2\}\rangle,\#,1) \subset \{1,\ldots,11\}$	(DefnUnfold-B L_{17})
L_6.	$\vdash 1 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(OrbitMember-B)
	$\vdash \cdots$	
L_{14}.	$\vdash 9 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(OrbitMember-B)
L_{15}.	$\vdash 10 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(OrbitMember-B)
L_{16}.	$\vdash 11 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(OrbitMember-B)
L_5.	$\vdash 1 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1) \wedge \ldots$	($\wedge$I-B L_6 $\ldots L_{16}$)
	$\wedge\, 11 \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	
L_4.	$\vdash \forall x{:}\{1,\ldots,11\}\blacksquare x \in Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	($\forall$I-FiniteSort-B L_5)
L_2.	$\vdash \{1,\ldots,11\} \subset Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(DefnUnfold-B L_4)
L_1.	$\vdash mv_O \doteq Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	(SubSetEqual-B L_2 L_3)
Thm.	$\vdash \exists O\blacksquare O \doteq Orbit(\langle\{a_1,a_2\}\rangle,\#,1)$	($\exists$I-B L_1)
	$a_1 = \{(1,10),(2,8),(11,3),(5,7)\},\ a_2 = \{(1,4,7,6),(10,9,2,11)\}$	

Figure 10.1: Orbit proof.

An Example

We exemplify the approach for the problem to determine (and prove) the orbit of 1 under the permutation group $M{=}\langle(1,10)(2,8)(3,11)(5,7),(1,4,7,6)(2,11,10,9)\rangle$. Figure 10.1 contains the $\mathcal{PDS}$ that is created at the highest level of abstraction. The problem of computing the concrete set, which is the orbit, is formalized via existential quantification given in line Thm. The first method applied introduces a meta-variable mv_O. InstPermTHFromGap introduces for this meta-variable the binding $mv_O{:=}^b\{1,2,3,4,5,6,7,8,9,10,11\}$ into the strategic proof plan. The rest of the proof is then to show that $\{1,2,3,4,5,6,7,8,9,10,11\}$ equals the orbit by double inclusion. The first direction, given in line L_2, is to show that all the points of the computed set are included in the orbit. The reverse inclusion in L_3 is closed by a fixed-point argument. It suffices to show that 1 is in the set, and the set is invariant for the generators of G.

L_{25}.	$\vdash mv_p \in \langle\{a_1,a_2\}\rangle$	(PermInGroup-B)
L_{27}.	$\vdash 9 \doteq 9$	($\doteq$Reflex-B)
L_{26}.	$\vdash 9 \doteq mv_p\#1$	(EvalPermutation-B L_{27})
L_{24}.	$\vdash \exists p{:}\langle\{a_1,a_2\}\rangle\mathbf{.}\, 9 \doteq p\#1$	($\exists$ISort-B L_{25} L_{26})
L_{14}.	$\vdash 9 \in \mathit{Orbit}(\langle\{a_1,a_2\}\rangle,\#,1)$	(DefnUnfold-B L_{24})

Figure 10.2: Expansion of OrbitMember-B.

L_{14} is justified by the unreliable method OrbitMember-B. Figure 10.2 gives the $\mathcal{PDS}$ segment that is constructed from PermStrat when this step becomes expanded and L_{14} becomes open again. The witness permutation, which maps 1 to 9 is introduced as meta-variable mv_p and bound by InstPermTHFromGap to $\{(1,9,2,8,11,3,10,4,7,5,6)\}$.

L_{31}.	$\vdash a_1 \in \langle\{a_1,a_2\}\rangle$	(InSet-B)
L_{30}.	$\vdash a_2 \in \langle\{a_1,a_2\}\rangle$	(InSet-B)
L_{29}.	$\vdash a_2 \circ a_1 \in \langle\{a_1,a_2\}\rangle$	(ProductOfGenerators-B L_{31} L_{30})
L_{28}.	$\vdash \{(1,9,2,8,11,3,10,4,7,5,6)\} \doteq a_2 \circ a_1$	(EqualWithGap-B)
L_{25}.	$\vdash \{(1,9,2,8,11,3,10,4,7,5,6)\} \in \langle\{a_1,a_2\}\rangle$	(RePresentWithGenerators-B L_{28} L_{29})
	$a_1 = \{(1,10),(2,8),(11,3),(5,7)\}$, $a_2 = \{(1,4,7,6),(10,9,2,11)\}$	

Figure 10.3: Expansion of PermInGroup-B.

This proof segment contains again an unreliable method application, namely L_{25} is justified by the unreliable method PermInGroup-B. The expansion of this step is given in Figure 10.3. PermStrat rewrites the permutation as a product of the generators. Then, the method EqualWithGap-B calls GAP to justify the equality of the permutations.

Whereas the $\mathcal{PDS}$ for the example has 22 lines on the most abstract level, the expansion of all unreliable method applications leads to a proof with 166 lines.

10.2 Interactive Theorem Proving with Multi

The homomorphism domain consists of problems involving the homomorphism property. Proof plans for homomorphism problems are constructed with the strategy HomStrategy. Although HomStrategy can solve homomorphism problems automatically our main focus was to tackle this domain interactively with Multi. This was motivated by the idea to integrate proof planning with this domain into a tutoring environment for an interactive mathematical course in algebra. The realized interactive proof planning benefits from Multi's flexible employment of different strategies. In particular, we exploit the strategy level in the tutor scenario to enable the flexible instantiation of meta-variables and the flexible deletion of steps.

We start with an introduction of the homomorphism domain. Then, we briefly discuss HomStrategy and how it tackles homomorphism problems. Afterwards, we motivate MULTI's tutor mode and illustrate it with an example from the homomorphism domain. A more detailed description of the use of MULTI in a tutoring environment can be found in [195].

Homomorphism Problems

The problems in the homomorphism domain range from standard problems concerning the homomorphism property as they can be found in standard mathematical textbooks on algebra such as [233] up to complex problems taken from [71]. As examples for both categories consider the following two problems:

1. $[\mathit{Group}(G,\circ) \wedge \mathit{Group}(H,\star) \wedge \mathit{Hom}(h,(G,\circ),(H,\star))]$
   $\Rightarrow \exists e{:}_{\mathit{Im}(h,G)}.\, \mathit{Unit}(\mathit{Im}(h,G),\star,e)$
2. $[\mathit{Group}(G,\circ) \wedge \mathit{Group}(H,\star) \wedge \mathit{Hom}(h_1,(G,\circ),(H,\star)) \wedge \mathit{Surj}(h_1,G,H)$
   $\wedge \mathit{Hom}(h_2,(G,\circ),(K,\diamond)) \wedge [\forall x{:}_G.\, h_2(x) \doteq \varphi(h_1(x))]]$
   $\Rightarrow [\mathit{Hom}(\varphi,(H,\circ),(K,\diamond))]$

The first problem states that, given a homomorphism h between two groups G and H, the image of h with respect to G contains a unit element. The second theorem, which is the most difficult of our homomorphism problems, states that if there are two groups G, H and a surjective homomorphism $h_1 : G \rightarrow H$ and if there is an additional homomorphism h_2 from G into some arbitrary structure $(K,\diamond)$ and a mapping $\varphi : H \rightarrow K$, such that $h_2(x) \doteq \varphi(h_1(x))$ for all $x \in G$, then φ is also a homomorphism.

Formalization

Some concepts relevant for the homomorphism domain are already introduced in section 5.2.2, for instance, homomorphism Hom, injectivity Inj, surjectivity Surj. The concepts Group, image Im, and kernel Kern are defined in ΩMEGA's database as follows:

$$\begin{aligned}
\mathit{Group}_{(\beta o)(\beta\beta\beta)o} &\equiv \lambda G_{\beta o}.\, \lambda \circ_{\beta\beta\beta}.\, \mathit{Closed}(G,\circ) \wedge \mathit{Assoc}(G,\circ) \\
&\qquad \wedge \exists e_\beta{:}_G.\, (\mathit{Unit}(G,\circ,e) \wedge \mathit{Inverse}(G,\circ,e)) \\
\mathit{Im}_{(\alpha\beta)(\alpha o)\beta o} &\equiv \lambda f_{\alpha\beta}.\, \lambda A_{\alpha o}.\, \lambda y_\beta.\, \exists x_\alpha{:}_A.\, y \doteq f(x) \\
\mathit{Kern}_{(\alpha\beta)(\alpha o)\beta\alpha o} &\equiv \lambda f_{\alpha\beta}.\, \lambda A_{\alpha o}.\, \lambda y_\beta.\, \lambda x_\alpha.\, [x \in A] \wedge [f(x) \doteq y]
\end{aligned}$$

Note that the image of a mapping f with respect to a set A is a subset of the codomain of f (i.e., the term $\mathit{Im}(f_{\alpha\beta}, A_{\alpha o})$ has the type βo). The kernel of a mapping f with respect to a set A and an element y from the codomain is a subset of the domain of f (i.e., the term $\mathit{Kern}(f_{\alpha\beta}, A_{\alpha o}, y_\beta)$ has the type αo). The concepts Closed, Assoc, Unit, and Inverse used here to formalize Group are also introduced and explained in section 5.2.2.

The HomStrategy

The basic approach of HomStrategy is to first unfold all definitions up to a point where the homomorphism property can be applied as often as possible; that is, if there is a homomorphism $h : A \rightarrow B$ HomStrategy tries to transform problems stated for elements of B into equivalent problems on A. Then, the proofs are concluded by deriving the necessary properties from the definition of A.

The central method in HomStrategy is APPLYHOM-B, which applies a homomorphism h backwards. That is, the application of APPLYHOM-B reduces a line-task with goal $\Phi[b_1 \circ b_2]$ and a support $\mathit{Hom}(h,(A,\star),(B,\circ))$ to the five new goals

$\Phi[h(mv_1 \star mv_2)]$, $h(mv_1)\dot{=}b_1$, $h(mv_2)\dot{=}b_2$, $mv_1 \in A$, and $mv_2 \in A$ where mv_1 and mv_2 are new meta-variables.

Occurring meta-variables are not instantiated by external systems but are bound by domain-specific methods that use and apply particular properties of groups. For instance, the methods UNITINGROUP-B and APPLYUNITGROUP-B rely on the existence of a unit element in a group. UNITINGROUP-B closes goals of the form $t \in G$ when there is a support $Group(G, \circ)$ and if t is either $groupunit(G, \circ)$ or a meta-variable. If t is a meta-variable mv, then the application of UNITINGROUP-B binds mv to $groupunit(G, \circ)$. APPLYUNITGROUP-B reduces an equation $t \circ d\dot{=}d$ or $d \circ t\dot{=}d$ to $d \in G$ when there is a support $Group(G, \circ)$ and if t is either $groupunit(G, \circ)$ or a meta-variable. If t is a meta-variable mv, then the application of APPLYUNITGROUP-B binds mv to $groupunit(G, \circ)$. INVERSEINGROUP-B and APPLYINVERSEGROUP-B are similar domain-specific methods in HomStrategy that rely on the inverse property.

Interactive Theorem Proving with MULTI

In the tutor scenario, a user should learn with MULTI how to tackle problems from a certain domain with methods that encode the typical steps in this domain. The user should be able to apply these methods flexibly and to combine the application of methods with meta-variable instantiation and the deletion of steps.

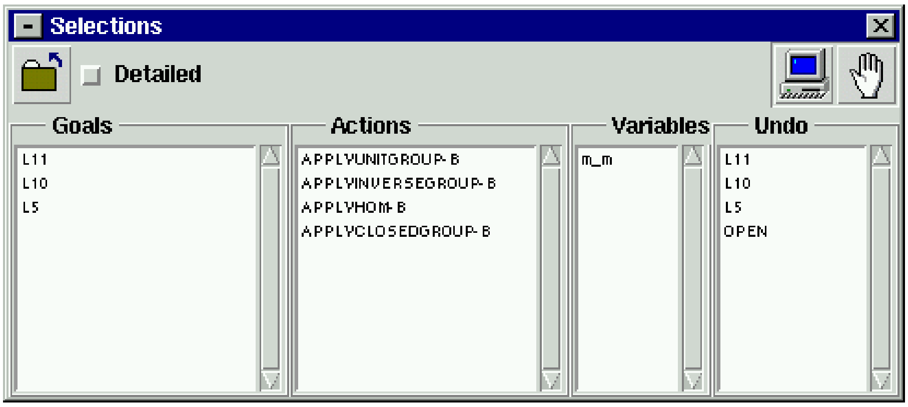
Our first approach to use MULTI for interactive proof construction was to integrate MULTI with ΩMEGA's user interface $\mathcal{L}\Omega\mathcal{UI}$. In this interactive mode the user can control each choice point in MULTI and its algorithms via $\mathcal{L}\Omega\mathcal{UI}$ (e.g., selecting the next strategy, the next task, the next method, the next supports, the next parameters, etc.). However, it turned out that this approach is not sufficient for a tutoring environment. The concrete control of the choice points in MULTI is possible only for an experienced user who has profound knowledge of MULTI and its algorithms. A user of a tutoring system cannot be expected to have this deep knowledge of the underlying system.

To overcome these problems we decided to hide the technical issues of MULTI and proof planning as much as possible. The user should be able to apply methods as well as to instantiate meta-variables and to perform backtracking but without noticing the technical details such as strategy and algorithm switching. Moreover, since the selection of suitable supports and parameters is often a painstaking effort the user should be supported here. We realized these ideas in a special mode of MULTI, which we call the *tutor mode*.

MULTI's Tutor Mode

When MULTI is invoked in tutor mode it obtains one **PPLANNER** strategy as argument that contains the methods whose application should be teached. We call this strategy the *tutor strategy*. MULTI invokes directly the tutor strategy on the initial line-task (provided that the application condition of the tutor strategy is satisfied by the initial line-task) such that the user is not confronted with the strategy level.

The communication between the user and MULTI in the tutor mode is realized via a special console that is integrated into $\mathcal{L}\Omega\mathcal{UI}$. The console pops up as soon as MULTI starts the tutor strategy. Figure 10.4 shows this console during the application of MULTI in tutor mode to the problem that $\exists e{:}_{Im(h,G)}.Unit(Im(h,G), \star, e)$ follows from $Group(G, \circ)$, $Group(H, \star)$ and $Hom(h,(G, \circ),(H, \star))$. Figure 10.5 contains the $\mathcal{PDS}$ at the moment, when the screen shot of the console was taken. Note that m_m is a meta-variable, which is displayed in the console as m_m.



Goals	Actions	Variables	Undo
L11	APPLYUNITGROUP-B	m_m	L11
L10	APPLYINVERSEGROUP-B		L10
L5	APPLYHOM-B		L5
	APPLYCLOSEDGROUP-B		OPEN

Figure 10.4: Operation console of MULTI in tutor mode.

$L_2.$	L_2	$\vdash Group(G,\circ)$	(Hyp)
$L_3.$	L_3	$\vdash Group(H,\star)$	(Hyp)
$L_4.$	L_4	$\vdash Hom(h,(G,\circ),(H,\star))$	(Hyp)
$L_5.$	$\mathcal{H}_1$	$\vdash m_m \in Im(h,G)$	$(Open)$
$L_8.$	L_8	$\vdash c \in Im(h,G)$	(Hyp)
$L_{10}.$	$\mathcal{H}_2$	$\vdash c \star m_m \doteq c$	$(Open)$
$L_{11}.$	$\mathcal{H}_2$	$\vdash m_m \star c \doteq c$	$(Open)$
$L_9.$	$\mathcal{H}_2$	$\vdash c \star m_m \doteq c \land m_m \star c \doteq c$	($\land$I-B L_{11} L_{10})
$L_7.$	$\mathcal{H}_1$	$\vdash \forall x{:}_{Im(h,G)} \centerdot x \star m_m \doteq x \land m_m \star x \doteq x$	($\forall$ISORT-B L_9)
$L_6.$	$\mathcal{H}_1$	$\vdash Unit(Im(h,G),\star,m_m)$	(DEFNUNFOLD-B L_7)
$L_1.$	$\mathcal{H}_1$	$\vdash \exists e{:}_{Im(h,G)} \centerdot Unit(Im(h,G),\star,e)$	($\exists$ISORT-B L_5 L_6)
$\mathcal{H}_1 = \{L_1,L_2,L_3\}$, $\mathcal{H}_2 = \{L_1,L_2,L_3,L_8\}$			

Figure 10.5: Homomorphism problem.

The console consists of four columns with entries and two fields with special symbols, namely a computer symbol and a hand symbol. The first column with the title *Goals* contains the current open lines. The second column whose title is *Actions* contains a subset of the methods of the tutor strategy. The entries of the third column with the title *Variables* are the current meta-variables whereas the fourth column with the title *Undo* contains again the current goals. The columns and the special fields correspond to choices of the user about the next proof manipulation to perform. We shall explain all possibilities in detail in the following. In general, it is important to note that the user does not have to follow the choice point sequence in **PPLANNER**. Rather the user can select entries in the console in an arbitrary order. Technically, this causes flexible jumps in the **PPLANNER** algorithm from one choice point to another choice point (also back to prior choice points).

The console restricts the choices of the user in the **PPLANNER** algorithm to task selection and action selection. The user selects a line-task by clicking the goal of the line-task in the first column. Then, the tutor strategy computes actions for this task and suggests them to the user in the second column (in the console in Figure 10.4 the user did click L_{11} such that the entries in the second column correspond to actions computed for the task with goal L_{11}). The computed actions are abbreviated in the second column by the name of their methods. When the user clicks an entry of the second column, then an additional window pops up in which the user can choose among different actions of the selected method (e.g., with different supports or parameters).

We could employ the action computation algorithm **CHOOSEACTIONALL** (see section 6.1.3) to compute the actions for a task. However, in tutor mode MULTI employs the ΩANTS mechanism (independent from the action computation algorithm of the tutor strategy). For each method in the tutor strategy there exists an ΩANTS agent[4] that computes actions for this method. We decided to use ΩANTS instead of **CHOOSEACTIONALL** for the action computation since it checks the methods concurrently. This provides an anytime character, so that the user can continue when a suitable action shows up and does not have to wait until all possible actions are computed. Moreover, it is possible to specify agents that create wrong suggestions, i.e., actions that are not applicable. This provides the independence to make wrong suggestions for pedagogical purposes in order to make the user find out what is wrong.

The user can also decide to instantiate occurring meta-variables and to delete steps. To instantiate a meta-variable the user clicks on the name of the meta-variable in the third column. Then, an additional window pops up with an input field in which the user can enter the desired instantiation. To delete steps the user clicks on an open line in the fourth column. This causes the deletion of the step that introduced the open line (and all steps that may depend from it). Technically, both operations are realized by strategy switches. The click of a meta-variable causes the switch from the tutor strategy to the **INSTMETA** strategy InstByUser. The instantiation computation function of InstByUser consists of a communication protocol that pops up the additional window and asks the user for an instantiation. The undo click causes a switch from the tutor strategy to the **BACKTRACK** strategy BackTrackActionToTask, which performs the desired backtracking.

Last but not least, the user can decide anytime to run the tutor strategy automatically and to return afterwards again to interactive proof development. The automated mode is invoked by a click on the field with the computer symbol, whereas it is interrupted again with a click on the field with the hand symbol. When the tutor strategy runs automatically, then it performs **PPLANNER**'s usual cycle of task selection, action selection, and action application. In particular, the action computation is performed by the computation algorithm of the strategy and not by ΩANTS agents.

We conclude the section with a short account on how to finish the problem in Figure 10.5. First, the user has to apply APPLYHOM-B to L_{10} and L_{11}, respectively. The application of APPLYHOM-B to L_{10} results — among others — in the goals $h(mv_1 \circ mv_2) \doteq h(mv_1)$ and $h(mv_2) \doteq m_m$. The former goal can be reduced to $mv_1 \circ mv_2 \doteq mv_1$, which can be closed with an application of APPLYUNITGROUP-B that binds mv_2 to $groupunit(G, \circ)$. The second goal is closed by $\doteq$REFLEX-B, which binds m_m to $h(groupunit(G, \circ))$. The goal L_{10} can be solved analog. It remains to prove in L_5 that $h(groupunit(G, \circ))$ is in $Im(h, G)$. To do so a $y \in G$ is needed such that $h(y) \doteq h(groupunit(G, \circ)$. A suitable y is $groupunit(G, \circ)$.

[4] The ΩANTS agents are not part of the **PPLANNER** strategy. Rather the agents relevant for the tutor strategy are identified directly from the methods of the tutor strategy (currently, corresponding agents and methods have the same name). Moreover, also the heuristics for ΩANTS are not part of the **PPLANNER** strategy. Rather, there is a fix set of ΩANTS heuristics that are employed for the tutor mode of MULTI.

Chapter 11

Conclusion and Outlook

This thesis presents proof planning with multiple strategies. Proof planning with multiple strategies is a novel approach extending proof planning by the new hierarchical level of strategies and their heuristic control in strategic control rules. The strategies are separate but collaborating operations, which can realize different plan refinements and modifications. The application of strategies is guided by meta-reasoning encoded in the strategic control rules that reason on the applicable strategies as well as on the whole proof planning status and the proof planning history. Both, the strategies and the strategic control rules can encode diverse (mathematical) domain knowledge beyond the capabilities of methods and method-level control rules.

We realized proof planning with multiple strategies in the Multi proof planner, which we implemented as a component of the ΩMEGA system. To enable the flexible combination of different strategies during a proof attempt Multi employs a blackboard architecture with two blackboards: the proof blackboard contains the status and the history of the proof planning problem, the control blackboard contains the information relevant for the control problem, that is, which possible step should the system perform next. We decided for a two-blackboard architecture to separate the control problem from the solution of the proof planning problem since both problems are equally important. The strategies are the knowledge sources that work on the proof blackboard. An invoked strategy can refine or modify the proof plan under construction and records its changes in a history. The knowledge source that works on the control blackboard is called the MetaReasoner. It evaluates the strategic control rules in order to prefer or reject the application of strategies.

We evaluated Multi with problems from several domains. In particular, we performed two large case studies in which we applied Multi to problems from the limit domain and problems of residue class structures. The case studies illustrate the domain knowledge at the strategy-level and and how it can be exploited for proof planning. In particular, we presented example problems that cannot be solved with the previous proof planner of ΩMEGA since their solution requires the flexible combination of different proof plan refinements. Multi can solve these problems and also all problems provable with the previous proof planner. Thereby, Multi benefits, in particular, from the meta-reasoning in strategic control rules that guide, for instance, the introduction of instantiations for variables or analyze failures to suggest particular plan refinements or modifications. Another major advantage of Multi that we exploit in the case studies is the realization of several proof techniques for one class of problems. This makes proof planning more robust: if one proof technique fails on a problem, another proof technique may solve it.

Possible Extensions

The modular structure of algorithms and strategies and the flexibility of MULTI's blackboard architecture ensure that necessary extensions can be easily realized. We discussed various possibilities to extend the multiple-strategy proof planning approach realized in MULTI throughout the thesis. In particular, the following extensions could be considered if there is a need for them.

Algorithms and Tasks MULTI is open for the integration of further algorithms that can contribute to the solution of a proof planning problem. Moreover, it is also possible to specify further kinds of tasks.

Concurrency Currently, MULTI employs no concurrency. However, concurrency could be beneficial at several points in MULTI. For instance, the applicability of strategies could be checked concurrently. This would avoid that a strategy whose applicability is difficult to check (which is not the case for the strategies currently employed) blocks MULTI. MULTI could continue as soon as some applicable strategies are found, rather than to wait until all applicability checks are done. Another possibility to employ concurrency could be the invocation of strategies. MULTI could invoke several promising strategies concurrently on several copies of a subproblem, rather than to decide for one strategy. This would allow to check the performance of several strategies on the concrete subproblem in a competitive manner.

Changing The Setting The user invokes MULTI with a set of strategies and a set of strategic control rules. Currently, MULTI cannot change afterwards the set of employed strategies or strategic control rules during its execution. To enable this, MULTI could place all control related issues on the control blackboard and allow for their manipulation by particular knowledge sources. For instance, MULTI could store all given strategies and strategic control rules on the control blackboard. The status of a strategy or a strategic control rule could be changed by knowledge sources from active to passive and vice versa. MULTI would then consider only active strategies for invocation and the MetaReasoner would evaluate only active control rules.

Goal-Directed Reasoning In general, the problem solving process in blackboard systems is event-driven, that is, knowledge sources are triggered by certain events. If the triggering events do not occur, then the knowledge source is not applicable and is not invoked. Goal-directed reasoning, in contrast, entails identifying and performing actions in order to perform and enable other actions, which may be desirable *per se* or because of their effects. We already employ some goal-directed reasoning in strategic control rules. More elaborate goal-directed reasoning could be realized with the construction and manipulation of meta-plans of desirable strategy invocations that guide the subsequent proof planning process: MULTI would try to invoke the next strategy of the meta-plan or, if this is not possible, it would try to invoke strategies that are likely to enable the next strategy in the meta-plan.

Availability

MULTI is implemented in Allegro Common Lisp with CLOS. It is available as part of the ΩMEGA system via the ΩMEGA home-page:

`http://www.ags.uni-sb.de/~omega`.

Appendix A

ChooseActionAll Algorithm

Input: (1) a task T, (2) a history $\vec{H}$, (3) a list of methods $\mathcal{M}$, (4) a list of control rules $\mathcal{C}$.

Output: Either a pair of an action and a list of actions or **fail**.

```
Algorithm: ChooseActionAll($T$,$\vec{H}$,M,C)
Let $T$=$L_{open} \blacktriangleleft \mathcal{SUPPS}_{L_{open}}$.

  1. Order Methods
     $Methods$:= evalcrules-methods($\mathcal{M}$,$\mathcal{C}$,$T$).
     Let $Methods = [M_1, \ldots, M_n]$.
     When $Methods$ empty then terminate and return fail.
     $Actions_1$:=initial-action-set($T, M_1$).
                   $\vdots$
     $Actions_n$:=initial-action-set($T, M_n$).

  2. Handle Task, Supports, Parameters, and Appl. Conditions
     For $i = 1$ to $n$:

     (a) Match Task Line
         Let $\ominus Concs_i$ the $\ominus$ conclusions of $M_i$.
         $Actions_i$:=match-task-line($L_{open}$,$\ominus Concs_i$,$Actions_i$).
     (b) Select and Match Supports and Parameters
         Let $\ominus Prems_i$ and $BPrems_i$ the $\ominus$ premises and blank premises of
         $M_i$. Let $Params_i$ the parameter variables of $M_i$.
         $Supps{+}Params_i$:=evalcrules-s+p($\mathcal{SUPPS}_{L_{open}}$,$\mathcal{C}$,$T$,$M_i$,$Actions_i$).
         $Actions_i$:= match-s+p($Supps{+}Params_i$,$\ominus Prems_i \cup BPrems_i$,
                                                   $Params_i$,$Actions_i$).
     (c) Evaluate Application Conditions
         $Actions_i$:=eval-appl-conds($Actions_i$,$M_i$).

     $Actions$:=$Actions_1 \cup \ldots \cup Actions_n$.
     When $Actions$ empty then terminate and return fail.
```

```
3. Outline Computations
   eval-outline-computations($Actions$).
   complete-outline($Actions$).

4. Choose Action
   $Actions$:=remove-backtracked($Actions$,$\vec{H}$).
   $Actions$:=evalcrules-actions($Actions$,$\mathcal{C}$).
   If $Actions = \emptyset$
      then
         Terminate and return fail.
      else
         Terminate and return first($Actions$).
```

Figure A.1: The **CHOOSEACTIONALL** algorithm.

Appendix B

Lim+ Example

Lim_f.	Lim_f	$\vdash \lim\limits_{x \to a} f(x) = l_f$	(Hyp)
Lim_g.	Lim_g	$\vdash \lim\limits_{x \to a} g(x) = l_g$	(Hyp)
L_2.	Lim_f	$\vdash \forall \epsilon_1 \blacksquare (0 < \epsilon_1 \Rightarrow \exists \delta_1 \blacksquare (0 < \delta_1 \land \forall x_1 \blacksquare (\lvert x_1 - a \rvert < \delta_1 \land \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < \epsilon_1)))$	(DefnUnfold-F Lim_f)
L_3.	Lim_g	$\vdash \forall \epsilon_2 \blacksquare (0 < \epsilon_2 \Rightarrow \exists \delta_2 \blacksquare (0 < \delta_2 \land \forall x_2 \blacksquare (\lvert x_2 - a \rvert < \delta_2 \land \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < \epsilon_2)))$	(DefnUnfold-F Lim_g)
L_{17}.	Lim_f	$\vdash 0 < mv_{\epsilon_1} \Rightarrow \exists \delta_1 \blacksquare (0 < \delta_1 \land \forall x_1 \blacksquare (\lvert x_1 - a \rvert < \delta_1 \land \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1}))$	($\forall$E-F L_2)
L_{18}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_1}$	(TellCS-B)
L_{20}.	$\mathcal{H}_3$	$\vdash \exists \delta_1 \blacksquare (0 < \delta_1 \land \forall x_1 \blacksquare (\lvert x_1 - a \rvert < \delta_1 \land \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1}))$	($\Rightarrow_E$ L_{18} L_{17})
L_{21}.	L_{21}	$\vdash 0 < c_{\delta_1} \land \forall x_1 \blacksquare (\lvert x_1 - a \rvert < c_{\delta_1} \land \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1})$	(Hyp)
L_{23}.	L_{21}	$\vdash 0 < c_{\delta_1}$	($\land$E-F L_{21})
L_{24}.	L_{21}	$\vdash \forall x_1 \blacksquare (\lvert x_1 - a \rvert < c_{\delta_1} \land \lvert x_1 - a \rvert > 0 \Rightarrow \lvert f(x_1) - l_f \rvert < mv_{\epsilon_1})$	($\land$E-F L_{21})
L_{25}.	L_{21}	$\vdash \lvert mv_{x_1} - a \rvert < c_{\delta_1} \land \lvert mv_{x_1} - a \rvert > 0 \Rightarrow \lvert f(mv_{x_1}) - l_f \rvert < mv_{\epsilon_1}))$	($\forall$E-F L_{24})
L_{38}.	Lim_g	$\vdash 0 < mv_{\epsilon_2} \Rightarrow \exists \delta_2 \blacksquare (0 < \delta_2 \land \forall x_2 \blacksquare (\lvert x_2 - a \rvert < \delta_2 \land \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2}))$	($\forall$E-F L_3)
L_{39}.	$\mathcal{H}_3$	$\vdash 0 < mv_{\epsilon_2}$	(TellCS-B)
L_{41}.	$\mathcal{H}_3$	$\vdash \exists \delta_2 \blacksquare (0 < \delta_2 \land \forall x_2 \blacksquare (\lvert x_2 - a \rvert < \delta_2 \land \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2}))$	($\Rightarrow_E$ L_{39} L_{38})
L_{42}.	L_{42}	$\vdash 0 < c_{\delta_2} \land \forall x_2 \blacksquare (\lvert x_2 - a \rvert < c_{\delta_2} \land \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2})$	(Hyp)
L_{44}.	L_{42}	$\vdash 0 < c_{\delta_2}$	($\land$E-F L_{42})
L_{45}.	L_{42}	$\vdash \forall x_2 \blacksquare (\lvert x_2 - a \rvert < c_{\delta_2} \land \lvert x_2 - a \rvert > 0 \Rightarrow \lvert g(x_2) - l_g \rvert < mv_{\epsilon_2})$	($\land$E-F L_{42})
L_{46}.	L_{42}	$\vdash \lvert mv_{x_2} - a \rvert < c_{\delta_2} \land \lvert mv_{x_2} - a \rvert > 0 \Rightarrow \lvert g(mv_{x_2}) - l_g \rvert < mv_{\epsilon_2}))$	($\forall$E-F L_{45})
L_{11}.	L_{11}	$\vdash \lvert c_x - a \rvert > 0 \land \lvert c_x - a \rvert < mv_\delta$	(Hyp)
L_{14}.	L_{11}	$\vdash \lvert c_x - a \rvert > 0$	($\land$E-F L_{11})
L_{13}.	L_{11}	$\vdash \lvert c_x - a \rvert < mv_\delta$	($\land$E-F L_{11})
L_5.	L_5	$\vdash 0 < c_\epsilon$	(Hyp)
L_{61}.	$\mathcal{H}_1$	$\vdash 0 \leq 0$	(AskCS-B)
L_{59}.	$\mathcal{H}_1$	$\vdash mv_\delta \leq c_{\delta_1}$	(TellCS-B)
L_{57}.	$\mathcal{H}_2$	$\vdash 0 \leq 0$	(AskCS-B)
L_{55}.	$\mathcal{H}_2$	$\vdash mv_\delta \leq c_{\delta_2}$	(TellCS-B)
L_{52}.	$\mathcal{H}_2$	$\vdash mv_{x_2} = c_x$	(TellCS-B)

L_{53}.	$\mathcal{H}_2$	$\vdash mv_{\epsilon_2} \leq \frac{1}{2} * c_\epsilon$	(TellCS-B)
L_{50}.	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2}$	(Solve*-B L_{13} L_{55})
L_{51}.	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert > 0$	(Solve*-B L_{14} L_{57})
L_{47}.	$\mathcal{H}_2$	$\vdash \lvert mv_{x_2} - a\rvert < c_{\delta_2} \wedge \lvert mv_{x_2} - a\rvert > 0$	($\wedge$I-B L_{50} L_{51})
L_{49}.	$\mathcal{H}_2$	$\vdash \lvert g(mv_{x_2}) - l_g\rvert < mv_{\epsilon_2}$	($\Rightarrow_E$ L_{47} L_{46})
L_{48}.	$\mathcal{H}_2$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	(Solve*-B L_{49} L_{52} L_{53})
L_{43}.	$\mathcal{H}_2$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\Rightarrow$E-F L_{47} L_{46} L_{48})
L_{40}.	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\exists$E-F L_{41} L_{43})
L_{37}.	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{1}{2} * c_\epsilon$	($\Rightarrow$E-F L_{39} L_{38} L_{40})
L_{31}.	$\mathcal{H}_1$	$\vdash \lvert 1\rvert \leq mv$	(TellCS-B)
L_{32}.	$\mathcal{H}_1$	$\vdash mv_{\epsilon_1} \leq \frac{c_\epsilon}{2*mv}$	(TellCS-B)
L_{33}.	$\mathcal{H}_1$	$\vdash \lvert g(c_x) - l_g\rvert < \frac{c_\epsilon}{2}$	(Simplify-B L_{37})
L_{34}.	$\mathcal{H}_1$	$\vdash 0 < mv$	(TellCS-B)
L_{35}.	$\mathcal{H}_1$	$\vdash mv_{x_1} = c_x$	(TellCS-B)
L_{29}.	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1}$	(Solve*-B L_{13} L_{59})
L_{30}.	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert > 0$	(Solve*-B L_{14} L_{61})
L_{26}.	$\mathcal{H}_1$	$\vdash \lvert mv_{x_1} - a\rvert < c_{\delta_1} \wedge \lvert mv_{x_1} - a\rvert > 0$	($\wedge$I-B L_{29} L_{30})
L_{28}.	$\mathcal{H}_1$	$\vdash \lvert f(mv_{x_1}) - l_f\rvert < mv_{\epsilon_1}$	($\Rightarrow_E$ L_{26} L_{25})
L_{27}.	$\mathcal{H}_1$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	(ComplexEstimate-B L_{28} L_{31} L_{32} L_{33} L_{34} L_{35})
L_{22}.	$\mathcal{H}_1$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\Rightarrow$E-F L_{26} L_{25} L_{27})
L_{19}.	$\mathcal{H}_3$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\exists$E-F L_{20} L_{21})
L_{16}.	$\mathcal{H}_3$	$\vdash \lvert((f(c_x) + g(c_x)) - l_f) - l_g\rvert < c_\epsilon$	($\Rightarrow$E-F L_{18} L_{17} L_{19})
L_{12}.	$\mathcal{H}_3$	$\vdash \lvert(f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$	(Simplify-B L_{16})
L_{10}.	$\mathcal{H}_4$	$\vdash \lvert c_x - a\rvert < mv_\delta \wedge \lvert c_x - a\rvert > 0$ $\Rightarrow \lvert(f(c_x) + g(c_x)) - (l_f + l_g)\rvert < c_\epsilon$	($\Rightarrow$I-B L_{12})
L_9.	$\mathcal{H}_4$	$\vdash \forall x{\scriptstyle\blacksquare}(\lvert x - a\rvert < mv_\delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon)$	($\forall$I-B L_{10})
L_8.	$\mathcal{H}_4$	$\vdash 0 < mv_\delta$	(TellCS-B)
L_7.	$\mathcal{H}_4$	$\vdash 0 < mv_\delta \wedge \forall x{\scriptstyle\blacksquare}(\lvert x - a\rvert < mv_\delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon)$	($\wedge$I-B L_8 L_9)
L_6.	$\mathcal{H}_4$	$\vdash \exists\delta{\scriptstyle\blacksquare}(0 < \delta \wedge \forall x{\scriptstyle\blacksquare}(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon))$	($\exists$I-B L_7)
L_4.	Lim_f, Lim_g	$\vdash 0 < c_\epsilon \Rightarrow \exists\delta{\scriptstyle\blacksquare}(0 < \delta \wedge$ $\forall x{\scriptstyle\blacksquare}(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < c_\epsilon))$	($\Rightarrow$I-B L_6)
L_1.	Lim_f, Lim_g	$\vdash \forall\epsilon{\scriptstyle\blacksquare}(0 < \epsilon \Rightarrow \exists\delta{\scriptstyle\blacksquare}(0 < \delta \wedge$ $\forall x{\scriptstyle\blacksquare}(\lvert x - a\rvert < \delta \wedge \lvert x - a\rvert > 0$ $\Rightarrow \lvert(f(x) + g(x)) - (l_f + l_g)\rvert < \epsilon)))$	($\forall$I-B L_4)
$LIM+$.	Lim_f, Lim_g	$\vdash \lim\limits_{x \to a}(f(x) + g(x)) = l_f + l_g$	(DefnUnfold-B L_1)

$\mathcal{H}_1 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}\}$, $\mathcal{H}_2 = \{Lim_f, Lim_g, L_5, L_{11}, L_{21}, L_{42}\}$
$\mathcal{H}_3 = \{Lim_f, Lim_g, L_5, L_{11}\}$, $\mathcal{H}_4 = \{Lim_f, Lim_g, L_5\}$

Appendix C

Limit Theorems

The following theorems from the limit domain can be proved by MULTI so far. We tested mainly conjectures from [12]. Many similar theorems could be formulated. In the following, X, Y denote sequences over the reals, f and g denote functions over the reals, and a, b denote arbitrary but fix reals. For problems marked with $(*)$ $\mathcal{CoSIE}$ fails to compute instantiations for meta-variables for the reasons discussed in section 8.2.3.

Limits of sequences

1. (Exercise 3.1.7 first part in [12])
   If the sequence $|X| = |(x_n)|$ has the limit 0, then the sequence $X = (x_n)$ has also the limit 0:
   $limseq\ |X| = 0 \Rightarrow limseq\ X = 0$

2. (Theorem 3.2.2 in [12])
   If the sequence $X = (x_n)$ has an limit l, then the sequence X is bounded:
   $limseq\ X = l \Rightarrow \exists m.\, 0 < m \wedge \forall n.\, |x_n| < m$

3. (Theorem 3.2.3.a first part in [12])
   If the sequence $X = (x_n)$ has the limit l_x and the sequence $Y = (y_n)$ has the limit l_y, then the sequence $X + Y = (x_n + y_n)$ has the limit $l_x + l_y$:
   $limseq\ X = l_x \wedge limseq\ Y = l_y \Rightarrow limseq\ X + Y = l_x + l_y$

4. (Theorem 3.2.3.a second part in [12])
   If the sequence $X = (x_n)$ has the limit l_x and the sequence $Y = (y_n)$ has the limit l_y, then the sequence $X - Y = (x_n - y_n)$ has the limit $l_x - l_y$:
   $limseq\ X = l_x \wedge limseq\ Y = l_y \Rightarrow limseq\ X - Y = l_x - l_y$

5. (Theorem 3.2.3.a third part in [12])
   If the sequence $X = (x_n)$ has the limit l_x and the sequence $Y = (y_n)$ has the limit l_y, then the sequence $X * Y = (x_n * y_n)$ has the limit $l_x * l_y$:
   $limseq\ X = l_x \wedge limseq\ Y = l_y \Rightarrow limseq\ X * Y = l_x * l_y$

6. (Theorem 3.2.3.a fourth part in [12])
   If the sequence $X = (x_n)$ has the limit l_x, then the sequence $a * X = (a * x_n)$ has the limit $a * l_x$:
   $limseq\ X = l_x \Rightarrow limseq\ a * X = a * l_x$

7. $(*)$(Theorem 3.2.3.b in [12])
   If the sequence $X = (x_n)$ has the limit l_x and the sequence $Y = (y_n)$ has the

limit $l_y \neq 0$ and $y_n \neq 0$ for all n, then the sequence $\frac{X}{Y} = (\frac{x_n}{y_n})$ has the limit $\frac{l_x}{l_y}$:
$limseq\ X = l_x \wedge limseq\ Y = l_y \wedge \forall n_{\blacksquare} y_n \neq 0 \Rightarrow limseq\ \frac{X}{Y} = \frac{l_x}{l_y}$

8. (Theorem 3.2.4 in [12])
If the sequence $X = (x_n)$ has a limit l and $x_n \geq 0$ for all n, then $l \geq 0$:
$limseq\ X = l \wedge \forall n_{\blacksquare} x_n \geq 0 \Rightarrow l \geq 0$

9. (Theorem 3.2.5 in [12])
If the sequence $X = (x_n)$ has a limit l_x and the sequence $Y = (y_n)$ has a limit l_y and $x_n \leq y_n$ for all n, then $l_x \leq l_y$:
$limseq\ X = l_x \wedge limseq\ Y = l_y \wedge \forall n_{\blacksquare} x_n \leq y_n \Rightarrow l_x \leq l_y$

10. (Theorem 3.2.6 in [12])
If the sequence $X = (x_n)$ has a limit l and $a \leq x_n \leq b$ for all n, then $a \leq l \leq b$:
$limseq\ X = l \wedge \forall n_{\blacksquare} a \leq x_n \leq b \Rightarrow a \leq l \leq b$

Limits of functions

1. (LIMC: Example 4.1.7.a in [12])
The function $f(x) = b$ has the limit b at a:
$\lim\limits_{x \to a} b = b$

2. (LIMV: Example 4.1.7.b in [12])
The function $f(x) = x$ has the limit a at a:
$\lim\limits_{x \to a} x = a$

3. (Example 4.1.7.c in [12])
The function $f(x) = x^2$ has the limit a^2 at a:
$\lim\limits_{x \to a} x^2 = a^2$

4. (∗) (LIM-DIV-1-X: Example 4.1.7.d in [12])
The function $f(x) = \frac{1}{x}$ has the limit $\frac{1}{a}$ at a, if $a > 0$:
$a > 0 \Rightarrow \lim\limits_{x \to a} \frac{1}{x} = \frac{1}{a}$

5. (∗) (Example 4.1.7.e in [12])
$\lim\limits_{x \to 2} \frac{x^3-4}{x^2+1} = \frac{4}{5}$

6. (Exercise 4.1.2 first part in [12])
If f has limit l at a, then the function $|f(x) - l|$ has the limit 0 at a:
$\lim\limits_{x \to a} f(x) = l \Rightarrow \lim\limits_{x \to a} |f(x) - l| = 0$

7. (Exercise 4.1.2 second part in [12])
If the function $|f(x) - l|$ has the limit 0 at a, then f has the limit l at a:
$\lim\limits_{x \to a} |f(x) - l| = 0 \Rightarrow \lim\limits_{x \to a} f(x) = l$

8. (Exercise 4.1.3 first part in [12])
If the function $f(x)$ has the limit l at a, then the function $f(x + a)$ has the limit l at 0:
$\lim\limits_{x \to a} f(x) = l \Rightarrow \lim\limits_{x \to 0} f(x + a) = l$

9. (Exercise 4.1.3 second part in [12])
If the function $f(x + a)$ has the limit l at 0, then the function $f(x)$ has the limit l at a:
$\lim\limits_{x \to 0} f(x + a) = l \Rightarrow \lim\limits_{x \to a} f(x) = l$

10. (Exercise 4.1.7 in [12])
If $k > 0$ and $|f(x) - l| \leq k * |x - a|$ for all x, then f has the limit l at a:
$k > 0 \wedge \forall x.\, |f(x) - l| \leq k * |x - a| \Rightarrow \lim\limits_{x \to a} f(x) = l$

11. (Exercise 4.1.8 in [12])
$\lim\limits_{x \to a} x^3 = a^3$

12. ($*$) (Exercise 4.1.10.a in [12])
$\lim\limits_{x \to 2} \frac{1}{1-x} = -1$

13. ($*$) (Exercise 4.1.10.b in [12])
$\lim\limits_{x \to 1} \frac{x}{1+x} = \frac{1}{2}$

14. ($*$) (Exercise 4.1.10.c in [12])
$\lim\limits_{x \to 0} \frac{x^2}{|x|} = 0$

15. ($*$) (Exercise 4.1.10.d in [12])
$\lim\limits_{x \to 1} \frac{x^2-x+1}{x+1} = \frac{1}{2}$

16. (Exercise 4.1.12 in [12])
If $f(x)$ has limit l at 0 and $a > 0$, then $f(a * x)$ has the limit l at 0:
$\lim\limits_{x \to 0} f(x) = l \wedge a > 0 \Rightarrow \lim\limits_{x \to 0} f(a * x) = l$

17. (Reverse of exercise 4.1.12)
If $f(a * x)$ has the limit l at 0 and $a > 0$, then $f(x)$ has limit l at 0:
$\lim\limits_{x \to 0} f(a * x) = l \wedge a > 0 \Rightarrow \lim\limits_{x \to 0} f(x) = l$

18. (Theorem 4.2.2 in [12])
If f has a limit at a, then f is bounded in a neighborhood of a:
$$\lim\limits_{x \to a} f(x) = l$$
$$\Rightarrow \exists m, \delta.\, m > 0 \wedge \delta > 0 \wedge \forall x.\, (|x - a| < \delta \wedge |x - a| > 0) \Rightarrow |f(x)| < m$$

19. (LIM+: Theorem 4.2.4.a first part in [12])
If f has limit l_f at a and g has limit l_g at a, then $f + g$ has limit $l_f + l_g$ at a:
$\lim\limits_{x \to a} f(x) = l_f \wedge \lim\limits_{x \to a} g(x) = l_g \Rightarrow \lim\limits_{x \to a} f(x) + g(x) = l_f + l_g$

20. (LIM-: Theorem 4.2.4.a second part in [12])
If f has limit l_f at a and g has limit l_g at a, then $f - g$ has limit $l_f - l_g$ at a:
$\lim\limits_{x \to a} f(x) = l_f \wedge \lim\limits_{x \to a} g(x) = l_g \Rightarrow \lim\limits_{x \to a} f(x) - g(x) = l_f - l_g$

21. (LIM*: Theorem 4.2.4.a third part in [12])
If f has limit l_f at a and g has limit l_g at a, then $f * g$ has limit $l_f * l_g$ at a:
$\lim\limits_{x \to a} f(x) = l_f \wedge \lim\limits_{x \to a} g(x) = l_g \Rightarrow \lim\limits_{x \to a} f(x) * g(x) = l_f * l_g$

22. (Theorem 4.2.4.a fourth part in [12])
If f has limit l_f at a, then $a * f$ has limit $a * l_f$ at a:
$\lim\limits_{x \to a} f(x) = l_f \Rightarrow \lim\limits_{x \to a} a * f(x) = a * l_f$

23. ($*$) (Theorem 4.2.4.b in [12])
If f has limit l_f at a and g has limit $l_g \neq 0$ at a and $g(x) \neq 0$ for all x, then $\frac{f}{g}$ has limit $\frac{l_f}{l_g}$ at a:
$\lim\limits_{x \to a} f(x) = l_f \wedge \lim\limits_{x \to a} g(x) = l_g \wedge \forall x.\, g(x) \neq 0 \Rightarrow \lim\limits_{x \to a} \frac{f(x)}{g(x)} = \frac{l_f}{l_g}$

24. (Example 4.2.5.b in [12])
$\lim\limits_{x \to 2}(x^2 + 1) * (x^3 - 4) = 20$

25. (Example 4.2.8.b in [12])
$\lim\limits_{x \to 0} sin(x) = 0$

26. (Example 4.2.8.c in [12])
$\lim\limits_{x \to 0} cos(x) = 1$

27. (Example 4.2.8.f in [12])
$\lim\limits_{x \to 0} x * sin(\frac{1}{x}) = 0$

28. (Exercise 4.2.1 in [12])
$\lim\limits_{x \to 1}(x + 1) * (2 * x + 3) = 10$

29. (Theorem 4.3.3 first part in [12])
If f has limit l at a, then f has the left-hand limit l at a:
$\lim\limits_{x \to a} f(x) = l \Rightarrow limL_{x \to a} f(x) = l$

30. (Theorem 4.3.3 second part in [12])
If f has limit l at a, then f has the right-hand limit l at a:
$\lim\limits_{x \to a} f(x) = l \Rightarrow limR_{x \to a} f(x) = l$

31. (Lim-If-Both-Sides-Lim: Theorem 4.3.3 third part in [12])
If f has the left-hand limit l and the right-hand limit l at a, then f has the limit l at a:
$limL_{x \to a} f(x) = l \land limR_{x \to a} f(x) = l \Rightarrow \lim\limits_{x \to a} f(x) = l$

Continuity of functions

1. (Example 5.1.5.a in [12])
The function $f(x) = b$ is continuous at a:
$cont(b, a)$

2. (Example 5.1.5.b in [12])
The function $f(x) = x$ is continuous at a:
$cont(x, a)$

3. (Example 5.1.5.b in [12])
The function $f(x) = x^2$ is continuous at a:
$cont(x^2, a)$

4. (Exercise 5.1.6 in [12])
If f is continuous at a, then for any $\epsilon > 0$ there exists a δ-neighborhood of a such that if x, y in this δ-neighborhood then $|f(x) - f(y)| < \epsilon$:
$$\begin{aligned} &cont(f, a) \Rightarrow \\ &\quad \forall \epsilon_{\blacksquare}(\epsilon > 0 \Rightarrow \exists \delta_{\blacksquare}(\delta > 0 \land \\ &\qquad\qquad \forall x, y_{\blacksquare}(|x - a| < \delta \land |y - a| < \delta \Rightarrow |f(x) - f(y)| < \epsilon))) \end{aligned}$$

5. (Exercise 5.1.11 in [12])
If $k > 0$ and $|f(x) - f(y)| \leq k * |x - y|$ for all x,y, then f is continuous at a:
$k > 0 \land \forall x, y_{\blacksquare} |f(x) - f(y)| \leq k * |x - y| \Rightarrow cont(f, a)$

6. (Continuous+: Theorem 5.2.1.a first part in [12])
If f is continuous at a and g is continuous at a, then $f + g$ is continuous at a:
$\mathit{cont}(f, a) \wedge \mathit{cont}(g, a) \Rightarrow \mathit{cont}(f + g, a)$

7. (Continuous-: Theorem 5.2.1.a second part in [12])
If f is continuous at a and g is continuous at a, then $f - g$ is continuous at a:
$\mathit{cont}(f, a) \wedge \mathit{cont}(g, a) \Rightarrow \mathit{cont}(f - g, a)$

8. (Continuous*: Theorem 5.2.1.a third part in [12])
If f is continuous at a and g is continuous at a, then $f * g$ is continuous at a:
$\mathit{cont}(f, a) \wedge \mathit{cont}(g, a) \Rightarrow \mathit{cont}(f * g, a)$

9. (Theorem 5.2.1.a fourth part in [12])
If f is continuous at a, then $a * f$ is continuous at a:
$\mathit{cont}(f, a) \Rightarrow \mathit{cont}(a * f, a)$

10. ($*$) (Theorem 5.2.1.b in [12])
If f is continuous at a and g is continuous at a and $g(x) \neq 0$ for all x, then $\frac{f}{g}$ is continuous at a:
$\mathit{cont}(f, a) \wedge \mathit{cont}(g, a) \wedge \forall x\ldotp g(x) \neq 0 \Rightarrow \mathit{cont}(\frac{f}{g}, a)$

11. (Theorem 5.2.7 in [12])
If f is continuous at a and g is continuous at $f(a)$, then the composition $g \circ f$ is continuous at a:
$\mathit{cont}(f, a) \wedge \mathit{cont}(g, f(a)) \Rightarrow \mathit{cont}(g \circ f, a)$

12. (Exercise 5.2.6 in [12])
If f has the limit l at a and g is continuous at l, then the composition $g \circ f$ has the limit $g(l)$ at a:
$\lim\limits_{x \to a} f(x) = l \wedge \mathit{cont}(g, l) \Rightarrow \lim\limits_{x \to a} g(f(x)) = g(l)$

13. (Cont-If-Lim=f)
If f has the limit $f(a)$ at a, then f is continuous at a:
$\lim\limits_{x \to a} f(x) = f(a) \Rightarrow \mathit{cont}(f, a)$

Derivatives of functions

1. ($*$) (Theorem 6.1.3.a in [12])
If f has the derivative f' at a, then $a * f$ has the derivative $a * f'$ at a:
$\mathit{deriv}(f, a) = f' \Rightarrow \mathit{deriv}(a * f, a) = a * f'$

2. ($*$) (Theorem 6.1.3.b in [12])
If f has the derivative f' at a and g has the derivative g' at a, then $f + g$ has the derivative $f' + g'$ at a:
$\mathit{deriv}(f, a) = f' \wedge \mathit{deriv}(g, a) = g' \Rightarrow \mathit{deriv}(f + g, a) = f' + g'$

3. ($*$) (Theorem 6.1.3.c in [12])
If f has the derivative f' at a and g has the derivative g' at a, then $f * g$ has the derivative $f' * g(a) + f(a) * g'$ at a:
$\mathit{deriv}(f, a) = f' \wedge \mathit{deriv}(g, a) = g' \Rightarrow \mathit{deriv}(f * g, a) = f' * g(a) + f(a) * g'$

4. ($*$) (Cont-If-Deriv: Theorem 6.1.2 in [12])
If f has a derivative at a, then f is continuous at a:
$\mathit{deriv}(f, a) = f' \Rightarrow \mathit{cont}(f, a)$

Bibliography

[1] A.J. Aho, J. Hopcroft, and J. Ullman, editors. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] J. Allen and D. Luckham. An Interactive Theorem-Proving Program. *Machine Intelligence*, 5:321–336, 1970.

[3] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The Nuprl Open Logical Environment. In McAllester [147], pages 170–176.

[4] P.B. Andrews. General Models and Extensionality. *The Journal of Symbolic Logic*, 37(2):395–397, 1972.

[5] P.B. Andrews. General Models, Descriptions and Choice in Type Theory. *The Journal of Symbolic Logic*, 37(2):385–394, 1972.

[6] P.B. Andrews. Transforming Matings into Natural Deduction Proofs. In Bibel and Kowalski [23], pages 281–292.

[7] P.B. Andrews. *An Introduction To Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, San Diego, CA, USA, 1986.

[8] P.B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[9] S. Autexier. *Hierarchical Contextual Rewriting.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2003. To appear.

[10] J. Avenhaus. *Reduktionssysteme.* Springer Verlag, Germany, 1995.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[12] R.G. Bartle and D.R. Sherbert. *Introduction to Real Analysis.* John Wiley& Sons, New York, 1982.

[13] P. Baumgartner and U. Furbach. PROTEIN, A PROver with a Theory INterface. In Bundy [40], pages 769–773.

[14] M. Beeson. Automatic generation of epsilon-delta proofs of continuity. In J. Calment and J. Plaza, editors, *Artificial Intelligence and Symbolic Computation*, pages 67–83. Springer Verlag, Germany, 1998.

[15] C. Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1999.

[16] C. Benzmüller, M. Bishop, and V. Sorge. Integrating TPS and OMEGA. *Journal of Universal Computer Science*, 5:188–207, 1999.

[17] C. Benzmüller, C. Brown, and M. Kohlhase. Higher order semantics and extensionality. *Journal of Symbolic Logic*, 2004. To appear.

[18] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMega: Towards a Mathematical Assistant. In McCune [151], pages 252–255.

[19] C. Benzmüller and M. Kohlhase. LEO – a Higher Order Theorem Prover. In Kirchner and Kirchner [136], pages 139–144.

[20] C. Benzmüller, A. Meier, and V. Sorge. Bridging Theorem Proving and Mathematical Knowledge Retrieval. In *Festschrift in Honour of Jörg Siekmann's 60s Birthday*. 2003. To appear.

[21] C. Benzmüller and V. Sorge. A Blackboard Architecture for Guiding Interactive Proofs. In Giunchiglia [101], pages 102–114.

[22] C. Benzmüller and V. Sorge. ΩANTS – An open approach at combining Interactive and Automated Theorem Proving. In Kerber and Kohlhase [134], pages 81–97.

[23] W. Bibel and R.A. Kowalski, editors. *Proceedings of the 5th Conference on Automated Deduction (CADE–5)*, volume 87 of *LNCS*, Les Arcs, France, June 7–9 1980. Springer Verlag, Germany.

[24] B. Bla and H. Geffner. Planning as Heuristic Search. *Journal of Artificial Intelligence*, 129(1–2):5–33, 2001.

[25] K.H. Bläsius and H.J. Bürckert, editors. *Deduktionssysteme*. Oldenbourg, 1992.

[26] W.W. Bledose and P. Bruell. A Man-Machine Theorem Proving System. In Nilsson [183], pages 56–65.

[27] W.W. Bledsoe. Some thoughts on proof discovery. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 2–10, 1986.

[28] W.W. Bledsoe. Challenge Problems in Elementary Analysis. *Journal of Automated Reasoning*, 6:341–359, 1990.

[29] W.W. Bledsoe, R.S. Boyer, and W.H. Henneman. Computer Proofs of Limit Theorems. *Artificial Intelligence*, 3(1):27–60, 1972.

[30] W.W. Bledsoe and P. Bruell. A Man-Machine Theorem Proving System. *Artificial Intelligence*, 5(1):51–72, 1974.

[31] W.W. Bledsoe and L. Hines. Variable Elimination and Chaining in a Resolution-Based Prover for Inequalities. In Bibel and Kowalski [23], pages 70 – 87.

[32] A. Blum and M.L. Furst. Fast Planning through Planning Graph Analysis. In C.S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642, Montreal, Canada, August 20–25 1995. Morgan Kaufmann, San Mateo, CA, USA.

[33] M.P. Bonacina. A taxonomy of theorem-proving strategies. In M.J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *LNAI*, pages 43–84. Springer Verlag, Germany, 1999.

[34] M.P. Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257, 2000.

[35] B. Buchanan, editor. *Proceedings of the 6th International Joint Conference on Artificial Intelligence (ICJAI)*, Tokyo, Japan, August 20–23 1979. Morgan Kaufmann.

[36] R. Bündgen. Application of the Knuth-Bendix Completion Algorithm to Finite Groups. Technical report, Univ. Tübingen, Germany, 1989.

[37] A. Bundy. Doing arithmetic with diagrams. In Nilsson [183], pages 130–138.

[38] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE–9)*, volume 310 of *LNCS*, pages 111–120, Argonne, Illinois, USA, 1988. Springer Verlag, Germany.

[39] A. Bundy. A science of reasoning. In *Computational Logic: Essays in Honor of Alan Robinson*. 1991.

[40] A. 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, Germany.

[41] A. Bundy. Proof Planning. In Drabble [73], pages 261–267.

[42] A. Bundy. A Critique of Proof Planning. In *Festschrift in Honour of Robort Kowalski*. 2002.

[43] A. Bundy, A. Stevens, F. van Hermelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[44] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[45] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam System. In Stickel [225], pages 647–648.

[46] A. Bundy, F. van Harmelen, A. Ireland, and A. Smaill. Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel [225], pages 132–146.

[47] J.G. Carbonell. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371–392. Morgan Kaufmann Publ., Los Altos, 1986.

[48] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[49] L. Cheikhrouhou and J. Siekmann. Planning Diagonalization Proofs. In Giunchiglia [101], pages 167–180.

[50] L. Cheikhrouhou and V. 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.

[51] S. Chien, S. Kambhampati, and C. Knoblock, editors. *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS2000)*, Breckenridge, CO, USA, April 15–17 2000. AAAI Press, Menlo Park, CA, USA.

[52] S. Choi and M. Kerber. Model-guided proof planning. *Logical and Computational Aspects of Model-Based Reasoning*, pages 143–162, 2002.

[53] S. Choi and A. Meier. Proof Planning in Omega with Semantic Guidance. Technical Report CSRP-01-11, University of Birmingham, School of Computer Science, 2001.

[54] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[55] L. Claesen and M. Gordon, editors. *Formal Methods in System Design — Special Issue on Higher Order Logic Theorem Proving and its Applications*, volume 3(1/2). Kluwer Academic Publisher, The Netherlands, 1993.

[56] W.J. Clancey and D. Weld, editors. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI-96)*, Portland, Oregon, USA, August 4–8 1996. AAAI Press, Menlo Park, CA, USA.

[57] A. Cohen, S. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In F. Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE–19)*, number 2741 in LNAI, Miami Beach, Fl, USA, 2003. Springer Verlag, Germany.

[58] S. Colton. *Automated Theory Formation in Pure Mathematics*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 2000.

[59] S. Colton. An Application-based Comparison of Automated Theory Formation and Inductive Logic Programming. *Linkoping Electronic Articles in Computer and Information Science (special issue: Proceedings of Machine Intelligence 17)*, 2002.

[60] S. Colton, A. Bundy, and T. Walsh. On the Notion of Interestingness in Automated Mathematical Discovery. *International Journal of Human Computer Studies*, 53(3):351–375, 2000.

[61] S. Colton, A. Bundy, and T. Walsh. Automatic Identification of Mathematical Concepts. In *Proceedings of the 17th International Conference on Machine Learning (ICML2000)*, pages 183–190. Morgan Kaufmann, USA, 2001.

[62] R.L. Constable, S.F. Allen, H.M. Bromley, R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[63] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 1999-2003. See `http://coq.inria.fr/doc/main.html`.

[64] D.D. Corkill, V.R. Lesser, and E. Hudlicka. Unifying Data-Directed and Goal-Directed Control. In D. Waltz, editor, *Proceedings of the Second National Conference on Artificial Intelligence (AAAI-82)*, pages 143 – 147, Carnegie-Mellon University / University of Pittsburgh, Pittsburgh, Pennsylvania, USA, August 18–20 1982. AAAI Press, Menlo Park, CA, USA.

[65] K. Currie and A. Tate. O-Plan: The Open Planning Architekture. *Artificial Intelligence*, 51(1):49–86, 1991.

[66] M. Davis. A computer program for Presburger's algorithm. In A. Robinson, editor, *Proving Theorems (as done by Man, Logician, or Machine)*, pages 215–233, 1957. Summary of talks presented at the 1957 summer institute for symbolic logic.

[67] M.D. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):394–397, July 1960.

[68] H. 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`.

[69] L. Dennis and A. Bundy. A Comparison of two Proof Critics: Power vs. Robustness. In V.A. Carreno, C.A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics: TPHOLs'02*, volume 2410 of *LNCS*, pages 182–197. Springer Verlag, Germany, 2002.

[70] J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10 – 15, Stockholm, Sweden, July 31–August 6 1999. Morgan Kaufmann, San Mateo, CA, USA.

[71] P. Deussen. *Halbgruppen und Automaten*, volume 99 of *Heidelberger Taschenbücher, Sammlung Informatik.* Springer Verlag, Germany, 1971.

[72] M.B. Do and S. Kambhampati. Solving Planning Graph by Compiling it into a CSP. In Chien et al. [51], pages 82–91.

[73] B. Drabble, editor. *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, Edinburgh, UK, May 29–31 1996. AAAI Press, Menlo Park, CA, USA.

[74] M. Drummond. On precondition achievement and the computational economics of automatic planning. In C. Bäckström and E. Sandwall, editors, *Current Trends in AI Planning.* IOS Press, 1994.

[75] E.H. Durfee and V.R. Lesser. Incremental Planning to Control a Blackboard-Based Problem Solver. In T. Kehler and S. Rosenschein, editors, *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 58 – 64, Philadelphia, Pennsylvania, USA, August 11–15 1986. AAAI Press, Menlo Park, CA, USA.

[76] R. Engelmore and T. Morgan, editors. *Blackboard Systems.* Addison-Wesley, 1988.

[77] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and R. Reddy. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 1980.

[78] L.D. Erman, P. London, and S. Fickas. The Design and an Example Use of HEARSAY-III. In Buchanan [35], pages 409–415.

[79] K. Erol, J. Hendler, and D. Nau. HTN Planning: Complexity and Expressivity. In B. Hayes-Roth and R.E. Korf, editors, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1123–1128, Seattle, Washington, USA, August 1–4 1994. AAAI Press, Menlo Park, CA, USA.

[80] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In Voronkov [235], pages 226 – 237.

[81] W.M. Farmer, J.D. Guttman, and F.J. Thayer. IMPS: System Description. In Kapur [130], pages 701 — 705.

[82] W.M. Farmer, J.D. Guttman, and F.J. Thayer. Little Theories. In Kapur [130], pages 567 — 581.

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

[84] A. Fiedler. *P.rex*: An Interactive Proof Explainer. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, volume 2083 of *LNAI*, Siena, Italy, June 18–22 2001. Springer Verlag, Germany.

[85] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Some New Directions in Robot Problem Solving. *Machine Intelligence*, 7, 1971.

[86] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[87] E. Fink. How to solve it automatically: Selection among problem-solving methods. In Simmons et al. [217], pages 128–136.

[88] M. Fox and D. Long. The automatic inference of State invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367 – 421, 1998.

[89] A. Franke and M. Kohlhase. System Description: MBase, an Open Mathematical Knowledge Base. In McAllester [147], pages 455–459.

[90] M. Fujita, J. Slaney, and F. Bennett. Automatic Generation of Some Results in Finite Algebra. In R. 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.

[91] H. 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, Germany.

[92] H. Ganzinger and U. Waldmann. Theorem Proving in Cancellative Abelian Monoids. In McRobbie and Slaney [157], pages 388–402.

[93] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4*, 1998.

[94] H. Gelernter. A Geometry-Theorem Proving Machine. In *Computers and Thought*, pages 134–152. McGraw Hill, 1963.

[95] H. Gelernter. Realization of a Geometry-Theorem Proving Machine. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning*, volume 1 Classical Papers on Computational Logic 1957–1966 of *Symbolic Computation*, pages 99–122. Springer Verlag, Germany, 1983.

[96] G. Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[97] S. Gerberding and A. Noltemeier. Incremental proof planning by meta-rules. In D.D. Dankel, editor, *Proceedings of the 10th International Florida Artificial Intelligence Research Symposium (FLAIRS-97)*, pages 171 – 175, Daytona Beach, Florida, USA, May 10–14 1997.

[98] S. Gerberding and B. Pientka. Structured Incremental Proof Planning. In O. Herzog and A. Günter, editors, *Advances in Artificial Intelligence, Proceedings of 22nd Annual German Conference on Artificial Intelligence*, volume 1504 of *LNAI*, Bremen, , Germany, September 15–17 1998. Springer Verlag, Berlin, , Germany.

[99] A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[100] M.L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25—46, 1993.

[101] F. Giunchiglia, editor. *Artificial Intelligence: Methodology, Systems and Applications, Proceedings of the of the 8th International Conference (AIMSA'98)*, volume 1480 of *LNAI*, Sozopol, Bulgaria, September 21–23 1998. Springer Verlag, Germany.

[102] F. Giunchiglia and T. Walsh. Theorem Proving with Definition. In *Proceedings of AISB 89, Society for the Study of Artificial Intelligence and Simulation of Behaviour*, 1989.

[103] C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24:67–100, 2000.

[104] C.P. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In C. Rich and J. 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.

[105] C.P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. In Simmons et al. [217], pages 208–213.

[106] M.J. Gordon, A. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, Germany, 1979.

[107] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, United Kingdom, 1993.

[108] P. Graf. *Term Indexing*. Number 1053 in LNCS. Springer Verlag, Germany, 1996.

[109] The Omega Group. $\mathcal{LOUI}$: $\mathcal{L}$ovely ΩMEGA $\mathcal{U}$ser $\mathcal{I}$nterface. *Formal Aspects of Computing*, 11:326–342, 1999.

[110] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In Chien et al. [51], pages 140–149.

[111] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 25:251–321, 1985.

[112] B. Hayes-Roth, F. Hayes-Roth, S. Rosenschein, and S. Cammarata. Modeling planning as an incremental, opportunistic process. In Buchanan [35], pages 375–383.

[113] L. Henkin. Completeness in the Theory of Types. *The Journal of Symbolic Logic*, 15:81–91, 1950.

[114] T. Hillenbrand, A. Jaeger, and B. Löchner. System Description: WALDMEISTER, Improvements in Performance and Ease of Use. In Ganzinger [91], pages 232 – 236.

[115] D.A. Hinsley, J.R. Hayes, and H.A. Simon. From words to equations: Meaning and representation in algebra word problems. In P.A. Carpenter and M.A. Just, editors, *Cognitive Processes in Comprehension*. Erlbaum, 1977.

[116] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253 – 302, 2001.

[117] A. Howe, E. Dahlman, C. Hansen, M. Scheetz, and A. von Mayrhauser. Exploiting Competitive Planner Performance. In S. Biundo and M. Fox, editors, *Recent Advances in AI Planning, Proceedings of the 5th European Conference on Planning (ECP'99)*, volume 1809 of *LNCS*, pages 62–72, Durham, UK, 1999. Springer Verlag, Germany.

[118] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. Ω-MKRP: A Proof Development Environment. In Bundy [40], pages 788–792.

[119] G.P. Huet. Experiments with an Interactive Prover for Logic with Equality. Technical report, Jennings Computing Center, Case Western Reserve University, 1970.

[120] G.P. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic*. PhD thesis, Case Western Reverse University, 1972.

[121] D. Hutter. Guiding inductive proofs. In Stickel [225].

[122] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov [235], pages 178 – 189.

[123] A. Ireland and A. Bundy. Productive Use of Failure in Inductive Proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[124] P. Jackson. Exploring Abstract Algebra in Constructive Type Theory. In Bundy [40], pages 590–604.

[125] A.K. Jonsson, P.H. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in Interplanetary Space: Theory and Practice. In Chien et al. [51], pages 177–186.

[126] M.V. Johnson Jr. and B. Hayes-Roth. Integrating Diverse Reasoning Methods in the BB1 Blackboard Control Architecture. In K. Forbus and H. Shrobe, editors, *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 30 – 35, Seattle, Washington, USA, July 13–17 1987. AAAI Press, Menlo Park, CA, USA.

[127] S. Kambhampati. Formalizing Dependency Directed Backtracking and Explanation-based Learning in Refinement Search. In Clancey and Weld [56], pages 757–762.

[128] S. Kambhampati. Refinement Planning: Status and Prospectus. In Clancey and Weld [56], pages 1331–1336.

[129] S. Kambhampati. Unifying Classical Planning Approaches. Technical Report ASU CSE TR 96-006, Arizona State University, 1996.

[130] D. 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, Germany.

[131] D. Kapur and D. Wang, editors. *Journal of Automated Reasoning— Special Issue on the Integration of Deduction and Symbolic Computation Systems*, volume 21(3). Kluwer Academic Publisher, The Netherlands, 1998.

[132] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Clancey and Weld [56], pages 1194–1201.

[133] H. Kautz and B. Selman. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In *Working notes of the AIPS-98-Workshop on Planning as Combinatorial Search*, 1998.

[134] M. Kerber and M. 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.

[135] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra Into Proof Planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[136] C. Kirchner and H. 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, Germany.

[137] H. Kirchner and C. Ringeissen, editors. *Proceedings of Third International Workshop on Frontiers of Combining Systems (FROCOS 2000)*, volume 1794 of *LNCS*, Nancy, France, March 22–24 2000. Springer Verlag, Germany.

[138] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[139] J. Köhler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning, Proceedings of the 4th European Conference on Planning (ECP'97)*, volume 1348 of *LNCS*, pages 273–285, Toulouse, France, 1997. Springer Verlag, Germany.

[140] M. Kohlhase. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1994.

[141] M. Kohlhase and A. Franke. MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems. *Journal of Symbolic Computation; Special Issue on the Integration of Computer algebra and Deduction Systems*, 32(4):365–402, 2001.

[142] R. Kowalski. A Proof Procedure Using Connection Graphs. *Journal of the ACM*, 22(4):572–595, 1975.

[143] T. Kropf, editor. *Formal Hardware Verification: Methods and Systems in Comparison*. Springer Verlag, Germany, 1997.

[144] U. Kühler. *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations*. PhD thesis, Sankt Augustin, 2000.

[145] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[146] D.W. Loveland, editor. *Automated Theorem Proving: A Logical Basis*. North-Holland, 1978.

[147] D. McAllester, editor. *Proceedings of the 17th International Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, Pittsburgh, PA, USA, June 17–20 2000. Springer Verlag, Germany.

[148] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In T.L. Dean and K. McKeown, editors, *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 634 – 639, Anaheim, California, USA, July 14–19 1991. AAAI Press, Menlo Park, CA, USA.

[149] W. 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.

[150] W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[151] W. 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, Germany.

[152] W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[153] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*, volume 1095 of *LNCS*. Springer Verlag, Germany, 1996.

[154] D. McDermott. A heuristic estimator for means-ends analysis in planning. In Drabble [73], pages 142–149.

[155] D. McDermott. PDDL – the planning domain definition language. 1998.

[156] D. McDermott. The 1998 AI Planning Systems Competition. *Artificial Intelligence Magazine*, 21(2):35–56, 2000.

[157] M.A. McRobbie and J.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, Germany.

[158] A. Meier. Randomization and heavy-tailed behavior in proof planning. Seki Report SR-00-03, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[159] A. Meier. Tramp: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In McAllester [147], pages 460–464.

[160] A. Meier, C. Gomes, and E. Melis. Randomization and restarts in proof planning. In A. Cesta and D. Borrajo, editors, *6th European Conference on Planning (ECP-01)*, LNCS. Springer, 2001.

[161] A. Meier, E. Melis, and M. Pollet. Towards Extending Domain Representations. Seki Report SR-02-01, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[162] A. Meier, M. Pollet, and V. Sorge. Exploring the Domain of Residue Classes. Seki Report SR-00-04, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[163] A. Meier, M. Pollet, and V. Sorge. Classifying Isomorphic Residue Classes. In R. Moreno-Díaz, B. Buchberger, and J.L. 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, Germany.

[164] A. Meier, M. Pollet, and V. Sorge. Classifying Residue Classes – Results of a Case Study. Seki Report SR-01-01, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[165] A. Meier, M. Pollet, and V. Sorge. Comparing Approaches to Explore the Domain of Residue Classes. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, 2002. S. Linton and R. Sebastiani, eds.

[166] A. Meier and V. Sorge. Exploring Properties of Residue Classes. In Kerber and Kohlhase [134], pages 175–190.

[167] A. Meier, V. Sorge, and S. Colton. Employing Theory Formation to Guide Proof Planning. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Proceedings of Joint International Conferences, AISC 2002 and Calculemus 2002*, pages 275 – 289, Marseille, France, 2002. Springer Verlag, Germany.

[168] E. Melis. AI-Techniques in Proof Planning. In H. Prade, editor, *Proceedings of of the 13th European Conference on Artifical Intelligence*, pages 494–498, Brighton, United Kingdom, August 23–28 1998. John Wiley & Sons, Chichester, United Kingdom.

[169] E. Melis. The "Limit" Domain. In Simmons et al. [217], pages 199–206.

[170] E. Melis and A. Bundy. Planning and Proof Planning. In S. Biundo, editor, *Proceedings of ECAI-96 Workshop on Cross-Fertilization in Planning*, pages 37–40, 1996.

[171] E. Melis and M. Pollet. Domain Knowledge for Search Heuristics in Proof Planning. In *Proceedings of AIPS-2000 Workshop: Analyzing and Exploiting Domain Knowledge*, pages 12–15, 2000.

[172] E. Melis and J. Siekmann. Knowledge-Based Proof Planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[173] E. Melis and C. Ullrich. Flexibly Interleaving Processes. In K.-D. Althoff and R. Bergmann, editors, *International Conference on Case-Based Reasoning*, volume 1650 of *LNAI*, pages 263–275. Springer Verlag, Germany, 1999.

[174] E. Melis, J. Zimmer, and T. Müller. Integrating Constraint Solving into Proof Planning. In Kirchner and Ringeissen [137], pages 32–46.

[175] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77–88. Prentice-Hall, 1984.

[176] S. Minton. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40:63–118, 1989.

[177] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[178] A. Newell, J.C. Shaw, and H.A. Simon. Empirical exploration with the logic theory machine. In *Proceedings of the Western Joint Computer Conference, Volume 15*, pages 218–239, 1957.

[179] A. Newell and H.A. Simon. GPS: a Program that Simulates Human Thought. In E.A. Feigenbaum and J. Feldmann, editors, *Computers and Thought*. McGraw-Hill, 1963.

[180] H.P. Nii, N. Aiello, and J. Rice. Frameworks for Concurrent Problem Solving: A Report on CAGE and POLIGON. In Engelmore and Morgan [76].

[181] H.P. Nii, E.A. Feigenbaum, J.J. Anton, and A.J. Rockmore. Signal-to-Symbol Transformation: HASP/SIAP Case Study. *AI Magazine*, 3(2):23–35, 1982.

[182] N. Nilsson. *Principles of Artificial Intellogence*. Tioga, 1980.

[183] N.J. Nilsson, editor. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Standford, CA, USA, August 20–23 1973. Morgan Kaufmann, San Mateo, CA, USA.

[184] A. Noltemeier. Inkrementelle Beweisplanung mit Metaregeln. Master's thesis, University of Darmstadt, 1996.

[185] H.S. Nwana and D.T. Ndumu. A Brief Introduction to Software Agent Technology. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 29 – 47. Springer Verlag, Germany, 1998.

[186] O-Plan-Team. *O-Plan Task Formalism (TF) Manual*. AI Applications Institute, University of Edinburgh, 1995.

[187] J. Otten and W. Bibel. leanCoP: Lean Connection-Based Theorem Proving. In P. Baumgartner and H. 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.

[188] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.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, Germany.

[189] L. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer Verlag, Germany, 1994.

[190] J. Pearl. *Heuristics.* Morgan Kaufmann, 1983.

[191] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114. Morgan Kaufmann, 1992.

[192] F. Pfenning. *Proof Transformations in Higher-Order Logic.* PhD thesis, Carnegie-Mellon University, Pittsburgh Pa., 1987.

[193] B. Pientka. Strukturierung der Beweisplanung mit Metaregeln. Master's thesis, University of Darmstadt, 1997.

[194] M.E. Pollack, D. Joslin, and M. Paolucci. Flaw Selection Strategies for Partial-Order Planning. *Journal of Artificial Intelligence Research*, 6:223–262, 1997.

[195] M. Pollet, E. Melis, and A. Meier. User interface for adaptive suggestions for interactive proof. In *In Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP 2003)*, Rome, Italy, 2003.

[196] G. Polya. *How to solve it.* Princeton University Press, 1971.

[197] A. Präcklein. The MKRP User Manual. Technical report, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1992.

[198] D. Prawitz. *Natural Deduction – A Proof-Theoretical Study.* Acta Universitatis Stockholmiensis 3. Almqvist & Wiksell, Stockholm, Sweden, 1965.

[199] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term Indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, pages 1853–1964. Elsevier Science and MIT Press, Cambridge, MA, USA, 2001.

[200] D. Redfern. *The Maple Handbook: Maple V Release 5.* Springer Verlag, Germany, 1999.

[201] J. 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.

[202] E. Rich and K. Knight, editors. *Artificial Intelligence.* McGraw-Hill, 1991.

[203] J. Richardson and A. Smail. Continuations of Proof Strategies. In R. Gor, A. Leitsch, and T. Nipkov, editors, *Short Papers of International Joint Conference on Automated Reasoning*, 2001.

[204] J.D.C. Richardson, A. Smaill, and I.M. Green. System description: Proof planning in higher-order logic with λ*Clam*. In Kirchner and Kirchner [136], pages 129–133.

[205] J.A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[206] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach.* Prentice Hall, Englewood Cliffs, 1995.

[207] E.D. Sacerdoti. The Non-Linear Nature of Plans. In C. Hewitt and P. Winston, editors, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, September 3–8 1975. Morgan Kaufmann, San Mateo, CA, USA.

[208] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNAI.* Springer Verlag, Germany, 1989.

[209] A.H. Schoenfeld. *Mathematical Problem Solving.* Academic Press, New York, 1985.

[210] S. Scholl. Hierarchische Analogie im Beweisplanen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbücken, 2003.

[211] J. Schumann. SiCoTHEO — Simple Competitive parallel Theorem Provers based on SETHEO. In *Proceedings of PPAI'95, Montreal, Canada*, 1995.

[212] J. Schumann and O. Ibens. *SETHEO V3.3 Reference Manual (Draft).* Institut für Informatik, TU München, 1997.

[213] J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof Development with OMEGA. In Voronkov [236], pages 144–149.

[214] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, and M. Pollet. Proof Development in OMEGA: The Irrationality of Square Root of 2. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series. Kluwer Academic Publishers, 2003. In Print.

[215] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof Development with OMEGA: Sqrt(2) is irrational. In M. Baaz and A. Voronkov, editors, *Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-2002)*, pages 367–387, Tbilisi, Georgia, 2002. Springer Verlag, Germany.

[216] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning — 1. Classical Papers on Computational Logic 1957 - 1966.* Springer Verlag, Germany, 1983.

[217] R. Simmons, M. Veloso, and S. Smith, editors. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Pittsburgh, PEN, USA, June 7–10 1998. AAAI Press, Menlo Park, CA, USA.

[218] J. Slaney. FINDER (Finite Domain Enumerator): Notes and Guide. Technical Report TR-ARP-1/92, Australian National University Automated Reasoning Project, Canberra, 1992.

[219] J. Slaney, M. Fujita, and M. Stickel. Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems. *Computers and Mathematics with Applications*, 29:115–132, 1995.

[220] S.F. Smith, O. Lassila, and M. Becker. Configurable, mixed-initiave systems for planning and scheduling. In A. Tate, editor, *Advanced Planning Technology.* AAAI Press, 1996.

[221] R.M. Smullyan. *First-Order Logic.* Springer Verlag, Germany, 1968.

[222] V. Sorge. Non-Trivial Symbolic Computations in Proof Planning. In Kirchner and Ringeissen [137], pages 121–135.

[223] V. Sorge. ΩANTS*: A Blackboard Architecture for the Integration of Reasoning Techniques into Proof Planning.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2001.

[224] R.M. Stallmann and G.J. Sussmann. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2), 1977.

[225] M. Stickel, editor. *Proceedings of the 10th International Conference on Automated Deduction (CADE–10)*, volume 449 of *LNAI*, Kaiserslautern, Germany, 1990.

[226] J. Stuber. Superposition Theorem Proving for Abelian Groups Represented as Integer Modules. *Theoretical Computer Science*, 208(1–2):149–177, 1998.

[227] M. Takahashi. Cut-Elimination in Simple Type Theory with Extensionality. *Journal of the Mathematical Society of Japan*, 19, 1968.

[228] M. 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.

[229] A. Tate. Generating Project Networks. In R. Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence (ICJAI)*, pages 888–893, Cambridge, MA, USA, August 22–25 1977. Morgan Kaufmann, San Mateo, CA, USA.

[230] Y. Tetsuya, A. Bundy, I. Green, T. Walsh, and D. Basin. Coloured rippling: An extension of a theorem proving heuristic. In A.G. Cohn, editor, *Proceedings of of the 11th European Conference on Artifical Intelligence*, pages 85 – 89. John Wiley & Sons, Chichester, United Kingdom, 1994.

[231] C. Ullrich. Analogie im Beweisplanen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbücken, 2000.

[232] 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.

[233] B.L. van der Waerden, editor. *Algebra.* Springer Verlag, Germany, 1966.

[234] M.M. Veloso, J. Carbonell, M.A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating Planning and Learning: The Prodigy Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[235] A. Voronkov, editor. *Proceedings of the 3rd International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNAI*, St. Petersburg, Russia, July 1992. Springer Verlag, Germany.

[236] A. Voronkov, editor. *Proceedings of the 18th International Conference on Automated Deduction (CADE–18)*, number 2392 in LNAI, Kopenhagen, Denmark, 2002. Springer Verlag, Germany.

[237] R. Waldinger. Achieving several goals simultaneously. *Machine Intelligence*, 8, 1977.

[238] H. Wang. Towards mechanical mathematics. *IBM Journal of Research and Development*, 4:2–22, 1960.

[239] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, Th. Engel, E. Keen, C. Theobalt, and D. Topic. System Description: Spass Version 1.0.0. In Ganzinger [91], pages 378–382.

[240] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER, Version 0.42. In McRobbie and Slaney [157], pages 141–145.

[241] D.S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 15(4):27–61, 1994.

[242] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4), 1990.

[243] D.E. Wilkins. Using the Sipe-2 Planning Systems (A Manual for Sipe-2, Version 6.1). Technical report, Stanford Research Institute (SRI), 2000.

[244] D.E. Wilkins and M. desJardins. A Call for Knowledge-based Planning. *Artificial Intelligence*, 22, 2001.

[245] D.E. Wilkins and K.L. Myers. A Multiagent Planning Architecture. In Simmons et al. [217], pages 154 – 162.

[246] D.E. Wilkins, K.L. Myers, J.D. Lowrance, and L.P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

[247] A. Wolf. Strategy Selection for Automated Theorem Proving. In Giunchiglia [101], pages 452 – 465.

[248] M.J. Wooldridge. Intelligent Agents. In G. Weiss, editor, *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*, pages 27–77. MIT Press, Cambridge, MA, USA, 1999.

[249] L. Wos. The Problem of Definition Expansion and Contraction. *Journal of Automated Reasoning*, 3:433–435, 1987.

[250] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, editors. *Automated Reasoning — Introduction and Applications.* Prentice Hall, 1984.

[251] H. Zhang. SATO: An Efficient Propositional Prover. In McCune [151], pages 272–275.

[252] H. Zhang, M. Bonacina, and H. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computations*, 21:543–560, 1996.

[253] J. Zhang and H. Zhang. Generating Models by SEM. In McRobbie and Slaney [157], pages 308–312.

[254] Z. Zilic and K. Radecka. On Feasible Multivariate Polynomial Interpolations over Arbitrary Fields. In S. 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.

[255] J. Zimmer. Constraintlösen für Beweisplanung. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2000.

[256] J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In Voronkov [236], pages 139–143.

[257] R. Zippel. Probabilistic Algorithms for Sparse Polynominals. In E.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, Germany.

[258] R. 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	25
$\mathcal{T}$	set of types	25
τ	type function	25
Σ	signature	25
$\neg$	negation	25
$\vee$	disjunction	25
Π^α	quantifier	25
ι^α	description operator	25
$\mathcal{V}$	set of variables	25
$(\mathbf{wff}_\alpha(\Sigma))_{\alpha\in\mathcal{T}}$	set of well formed formulas of α	26
$\mathbf{wff}(\Sigma)$	set of well-formed formulas over Σ	26
‘**.**’	λ separator	26
$\mathbf{FV}(\mathbf{A})$	set of free variables of $\mathbf{A}$	26
$\rightarrow_\alpha$	α-conversion	26
$\rightarrow_\beta$	β-reduction	26
$\rightarrow_\eta$	η-reduction	26
$\mathbb{N}$	set of non-negative integers	26
$\mathbb{N}^*$	set of all words over $\mathbb{N}$	26
ϵ	empty word	26
‘.’	concatenation of words	27
$\langle\pi\rangle$	term position π	27
$\top$	truth	27
$\bot$	falsehood	27
$\mathcal{I}$	interpretation of constants	27
$\forall$	universal quantifier	28
$\exists$	existential quantifier	28

$\wedge$	conjunction	28
$\Rightarrow$	implication	28
$\Leftrightarrow$	equivalence	28
$\doteq^{\alpha}$	equality	28
φ	variable assignment	28
$\mathcal{I}_{\varphi}$	denotation	28
$\mathbf{M} = \langle \mathcal{D}, \mathcal{I} \rangle$	Henkin model	28
$\models$	semantical consequence	29
$\mathcal{H} \vdash_{ND} F$	syntactical consequence	32
$\mathcal{H} \vdash F$	syntactical consequence	32
$\equiv$	definition symbol	34
$\mathcal{PDS}$	proof plan data structure	38
$[]$	Empty list	43
$\cap$	Intersection of sets	43
$\cup$	Union of sets,concatenation of lists	43
$\vec{A}$	sequence of actions	55
$\mathcal{P}$	$\mathcal{PDS}$	55
$\hat{\mathbf{A}}$	agenda	55
$\vec{H}$	history	55
Φ	action introduction function	56
$\vec{\Phi}$	recursive action introduction function	57
Φ^{-1}	action deletion function	60
lim	limit of functions	67
$cont$	continuity of functions	67
$limseq$	limit of sequence	67
$deriv$	derivative of function	67
$Closed$	Property of structures	72
$Assoc$	Property of structures	72
$Unit$	Property of structures	72
$Inverse$	Property of structures	72
$Divisors$	Property of structures	72
$Commu$	Property of structures	72
$Distrib$	Property of structures	72

Hom	Property of function between structures	73
Inj	Property of function between structures	73
$Surj$	Property of function between structures	73
Iso	Property of function between structures	73
$cl_n(m)$	congruence class m modulo n	73
RS	residue class set	73
$\bar{+}$	addition on congruence classes	73
$\bar{*}$	multiplication on congruence classes	73
$\bar{-}$	subtraction on congruence classes	73
$\mathcal{B}\mathbf{S}$	binding store	114
$\vec{\mathcal{B}\mathbf{S}}$	sequence of binding stores	114
$\Phi_{\textsc{Multi}}$	action introduction function	119
$\vec{\Phi}_{\textsc{Multi}}$	recursive action introduction function	119
$\mathbf{PB}$	proof blackboard	127
$\mathbf{CB}$	control blackboard	127
$\Phi^{-1}_{\textsc{Multi}}$	action deletion function	140
$\vec{\Phi}^{-1}_{\textsc{Multi}}$	recursive action deletion function	140
$Pair$	pairing function	187
$LProj$	projection for left element of pair	188
$RProj$	projection for right element of pair	188
$\otimes$	direct product of two sets	188
$\times$	operation on direct products	188
$Group$	Structure	221
Im	Set wrt. to function	221
$Kern$	Set wrt. to function	221

Index of Names

Index