Assertion Level Proof Planning with Compiled Strategies

Dominik Dietrich

Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes.
Saarbrücken, 2011.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Dietrich, Dominik:
Assertion Level Proof Planning with Compiled Strategies
ISBN 978-3-86376-015-1

Tag des Kolloquiums: 27. September 2011

Kurzzusammenfassung

Die vorliegende Arbeit beschäftigt sich damit, das Formalisieren von Beweisen zu vereinfachen, indem Methoden entwickelt werden, um informale Beweise formal zu verifizieren und erzeugen zu können. Dazu wird ein abstrakter Kalkül entwickelt, der direkt auf der Faktenebene arbeitet, welche von Menschen geführten Beweisen relativ nahe kommt. Anhand einer Fallstudie wird gezeigt, dass die abstrakte Beweisführung auf der Fakteneben vorteilhaft für automatische Suchverfahren ist. Zusätzlich wird eine Strategiesprache entwickelt, die es erlaubt, unterspezifizierte Beweismuster innerhalb des Beweisdokumentes zu spezifizieren und Beweisskizzen automatisch zu verfeinern. Fallstudien zeigen, dass komplexe Beweismuster kompakt in der entwickelten Strategiesprache spezifiziert werden können. Zusammen bilden die einander ergänzenden Methoden den Rahmen zur Automatisierung von deklarativen Beweisen auf der Faktenebene, die bisher überwiegend manuell entwickelt werden mussten.

Abstract

The objective of this thesis is to ease the formalization of proofs by being able to verify as well as to automatically construct abstract human-style proofs. This is achieved by lifting the logical basis to the abstract *assertion level*, which has been identified as a style of reasoning that can be found in textbooks. A case study shows that automatic reasoning procedures benefit from the abstract assertion level reasoning. In addition, a strategy language is developed that allows the specification of abstract underspecified declarative proof patterns within the proof document and supports their refinement. Case studies show that complex reasoning patterns can concisely be specified within the developed language. Together, the complementary methods provide a framework to automate declarative proofs at the assertion level.

Acknowledgements

First of all, I would like to thank Prof. Dr. Jörg Siekmann who accepted me as his Ph.D. student, and who has, over the past years, given me all the encouragement and conditions necessary to carry out this thesis. He has given me a lot of freedom to work on my thesis and his knowledge, interest in the research, and guidance has helped me to complete this thesis. My sincere gratitude goes to Prof. Dr. Alan Bundy, who with great experience in the field and his thorough understanding of my work engaged me in valuable discussions. I am grateful that he agreed to serve as an examiner of the thesis. I would also like to thank Prof. Dr. Fairouz Kamareddine for agreeing to become an examiner of this thesis. Moreover, I want to express my deeply-felt thanks to my thesis advisor Serge Autexier for his warm encouragement and thoughtful guidance during the entire period. This thesis greatly benefited from his scientific advice, including the more technical parts of this thesis.

My research visit at the Carnegie Mellon University was one of the most wonderful experiences during the time of my Ph.D. study. Therefore, my special thanks go to Prof. Wilfried Sieg for giving me this great opportunity to be a guest at his laboratory. I sincerely thank him for his time, discussions and hospitality, from which this thesis benefited enormously. Moreover, I want to thank the DAAD for the financial support.

The members of the ΩMEGA group in Saarbrücken and FormalSafe group at DFKI Bremen have contributed immensely to my personal and professional time. The groups have been a source of friendships as well as good advice and collaboration. In particular, I wish to mention Christoph Benzmüller, Mark Buckley, Dieter Hutter, Christian Maeder, Till Mossakowski, Martin Pollet, Marvin Schiller, Ewaryst Schulz, Lutz Schröder, Holger Täubig, Marc Wagner, Dennis Walter, and Claus-Peter Wirth.

I wish to thank all anonymous and known reviewers of my papers for providing thoughtful comments, from which this thesis greatly benefited, and all people who supported me during the time of writing this thesis. In particular I want to mention my colleague and friend Ewaryst Schulz for patiently proof-reading almost the complete thesis and for discussions about this work, as well as Christoph Benzmüller, who also carefully read many parts of this thesis. Moreover, I want to express my appreciation to Till Mossakowski, Lutz Schröder, and Dennis Walter for reading parts of the thesis.

Finally I would like to thank my parents for the financial support over the years, as well as my girlfriend Sandra for her love and patience.

Contents

List of Figures

List of Tables

Zusammenfassung

Formale Beweise werden zunehmend in praktischen Anwendungen eingesetzt, um beispielsweise die Korrektheit sicherheitskritischer Hard- und Softwarekomponenten oder allgemeiner mathematischer Sätze nachzuweisen. Trotz erheblicher Fortschritte in der Automatisierung formaler Beweise müssen diese in der Regel manuell, das heißt von einem Experten, erstellt werden. Da jeder einzelne Beweisschritt durch eine Kalkülregel abgedeckt werden muss, sind formale Beweise sehr umfangreich und aufwendig zu führen. Insbesondere werden sie unlesbar lang, was eine Präsentation des Beweises und seiner zugrunde liegenden Beweisidee sehr schwierig macht.

Die vorliegende Arbeit beschäftigt sich damit, das Formalisieren von Beweisen zu vereinfachen, indem Methoden entwickelt werden, um informale Beweise formal zu verifizieren und erzeugen zu können. Es werden Beweisdokumente entwickelt, die menschenles- und schreibbar sind, aber auch maschinell erzeugt und auf Korrektheit überprüft werden können. Zusätzlich wird das Spezifizieren von Beweisstrategien innerhalb des Beweisdokumentes unterstützt. Das wird durch folgende Techniken möglich:

Erstens: Wir entwickeln einen abstrakten Kalkül, mit dem Beweise direkt auf der sogenannten *Faktenebene* konstruiert und verifiziert werden können. Die Faktenebene ist ein Beweisstil, in dem jeder Beweisschritt einer Anwendung eines mathematischen Satzes, eines Axioms, oder einer Annahme entspricht, was Beweisen aus Textbüchern relativ nahe kommt. Verglichen mit Beweisen im Sequenzenkalkül erhält man eine Reduktion der Schrittanzahl um ungefähr eine Zehnerpotenz. Aus diesem Grund sind solche Beweise wesentlich kompakter und können menschenverständlich dargestellt werden. Obwohl die Faktenebene bereits vor über fünfzehn Jahren eingeführt wurde, war es bisher nicht möglich, Beweise direkt auf der Faktenebene zu konstruieren. Stattdessen konnten sie lediglich aus Resolutionsbeweisen oder Beweisen im Kalkül des natürlichen Schließens durch Beweistransformation und Abstraktion gewonnen werden. Um die Anzahl der Beweisschritte weiter zu reduzieren, kombinieren wir die Faktenebene mit der Technik des tiefen Schließens, die Schlussregeln derart erweitert, dass sie auch auf Teilformeln angewendet werden können, wie man es von Gleichungsanwendungen kennt. Dadurch können Beweise konstruiert werden, deren Länge exponentiell kürzer ist als Beweise im Sequenzenkalkül ohne Schnitt.

Zweitens: Wir entwickeln eine Strategiesprache, die es erlaubt, allgemeine Beweissuchverfahren und häufige Beweismuster innerhalb von Beweisdokumenten zu spezifizieren und somit Standardaufgaben effizient zu automatisieren. Die Sprache stellt Mechanismen bereit, um die Anwendung von Schlussregeln zu steuern und sie an konkrete Anwendung anzupassen. Außerdem führen wir den Begriff der deklarativen Taktik ein, die es erlaubt, unterspezifizierte deklarative Beweisskripte zu erzeugen und dabei ihre Granularität, Benennungen von wichtigen Voraussetzungen, oder die Wahl zwischen Vorwärts- und Rückwärsschritten zu steuern. Die Effizienz der Strategien wird durch Kompilierungstechniken erreicht.

Damit stellen wir insgesamt zwei Möglichkeiten zur Verfügung, sogenannte deklarative

Beweise zu automatisieren, die bisher überwiegend nur manuell entwickelt werden. Zum einen können Faktenbeweise in deklarative Beweisskripte umgewandelt werden. Zum anderen erzeugen deklarativen Taktiken direkt deklarative Beweisskripte und kombinieren somit die Vorteile des deklarativen und prozeduralen Beweisstils. Für die Beweisplanung stellen deklarative Strategien ein neues Speicherformat dar, das es ermöglicht, Beweismuster kompakt zu repräsentieren. Insbesondere unterstützen sie das abstrakte Suchen und anschließende Verfeinern von Beweisplänen. Dadurch, dass ein Großteil des Suchwissens deklarativ innerhalb des Beweisdokumentes spezifiziert werden kann, erhöht sich die Benutzbarkeit von Beweisplanern und es wird möglich, Beweisplanungssysteme zu evaluieren. Insbesondere erreichen wir eine Trennung der Strategiesprache von der Implementierungssprache, was die Wartbarkeit des Systems erhöht und zu effizienterem Laufzeitverhalten führt, weil Optimierungen durchgeführt werden können, die normalerweise zu aufwendig sind.

Die unterschiedlichen Aspekte dieser Arbeit wurden mit folgenden Ergebnissen evaluiert: (i) Die Problemklasse der Statmantautologien, deren Beweise $O(2^n)$ Schritte im Sequenzenkalkül benötigen, können effizient in $O(n^2)$ ohne Rücksetzungsschritte gelöst werden. (ii) Die Faktenebene erlaubt das Automatisieren von Testproblemen in der Mengentheorie aus der TPTP Bibliothek, die von vielen aktuellen Beweisern nicht gelöst werden können. Dabei kommen ausschließlich allgemeingültigen Suchstrategien zum Einsatz. Das Ergebnis zeigt, dass die Faktenebene nicht nur die Qualität der Beweise verbessert, sondern auch sehr nützlich für die eigentliche Beweissuche ist. (iii) Spezielle Suchalgorithmen auf der Faktenebene erlauben es, einen Korpus von tutoriellen Dialogen, die innerhalb des DIALOG-Projektes gesammelt wurden, korrekt abzubilden und zu klassifizieren. Die erforderliche Rekonstruktion löst unterspezifizierte und unvollständige Eingaben auf, was über reine Korrektheitsüberprüfung hinausgeht und nicht mit herkömmlichen Methoden erreicht werden kann. Die rekonstruierten Beweise bilden die Grundlage für eine weitere Analyse der Eingabe, wie beispielsweise einer Granularitätsanalyse, die nicht mit herkömmlichen Kalkülen durchgeführt werden kann. (iv) Wir automatisieren Beweise in der sogenannten Limesdomäne, welche sich mit Grenzwertsätzen beschäftigt. Verglichen mit einer vorherigen Referenzimplementierung wird gezeigt, dass die Heuristiken deutlich kompakter repräsentiert werden können, gemessen in Spezifikationszeilen der Strategien. Dies ist durch neue Kontrollflüsse möglich, die vorher nicht ausgedrückt werden konnten. Die Kompaktheit erhöht die Lesbarkeit und Wartbarkeit der Strategien und reduziert den Spezifikationsaufwand.

Insgesamt liefern die entwickelten Techniken wesentliche Beiträge im Bereich des interaktiven Beweisens und der Beweisplanung und eröffnen neue Anwendungsmöglichkeiten für automatische Beweiser in Bereichen, in denen die Nachvollziehbarkeit von Schritten wichtig ist, wie zum Beispiel in Lernumgebungen für mathematische Beweise. Das entwickelte System wurde erfolgreich im Bereich des interaktiven Beweisens sowie als Modul der mathematischen e-Learning Plattform ACTIVEMATH eingesetzt.

Extended Abstract

Formal proofs are increasingly employed in practical applications and used to verify critical pieces in hard- and software, as well as mathematics. Even though efficient automated reasoning procedures have been developed within the last decades and also been used to prove open conjectures, the mechanization of a proof is still an enormously difficult undertaking. This is because many proofs still need to be constructed interactively due to practical limitations of automated search procedures. Moreover, the level of detail of a formal proof is significantly higher than in human proofs, resulting in very large proofs. This so-called formal noise hides the line of reasoning that can be followed and understood by humans, and the intelligible presentation of formal proofs is usually not attempted.

The objective of this thesis is to ease the formalization of proofs by being able to verify as well as to automatically construct abstract human-style proofs. Our approach is centered around proof documents that are human-readable, human-writable, machine-checkable, machine-producible, and support the specifications of proof strategies within the proof document. We approach this goal from two angles.

First, we develop an abstract calculus that allows the construction of proofs directly at the *assertion level*, a proof style in which each reasoning step corresponds to the application of an axiom, lemma, theorem, or hypothesis. It has been identified as a proof style that comes close to proofs as written by humans. Due to their abstract nature, assertion level proofs substantially reduce the noise of formal proofs and lead to an understandable, i.e., human-readable, presentation. While being introduced already more than fifteen years ago, proofs could not directly be constructed at the assertion level, but must be obtained by transformation of an underlying proof in natural deduction or resolution. The ability to perform proof search directly at the assertion level thus identifies assertion level proofs to be directly machine-checkable and machine-producible. To further reduce the proof size, we combine the assertion level mechanism with the *deep inference* paradigm, which generalizes the notion of inference rules by allowing their application at arbitrary depth inside logical expressions, similar to rewriting rules. The increased freedom allows the construction of new proofs that are exponentially shorter and not available in the sequent calculus without cut.

Second, we develop a *strategy language* to express generic search strategies as well as common patterns of reasoning within the proof document with the objective to efficiently automate routine tasks. The language provides constructs to control the increased non-determinism which comes as a side-effect of deep inference and allows its adjustment according to the needs of the application given at hand. Moreover, it provides the notion of a *declarative tactic* as a means to produce underspecified, declarative proof scripts, thereby controlling stylistic choices such as their granularity, naming of important formulas that are expected to play a major role in the subsequent proof, or the choice between forward and backward steps. Compilation techniques are used to increase efficiency.

Our approach naturally provides two possibilities to automate the declarative style of proof, which has been established as the means of proof presentation of choice, but which is not sufficiently automated by current state of the art proof assistants. First, abstract assertion level proofs can directly be translated to declarative proof scripts. Second, we introduce the notion of a declarative tactic, which direclty produce declarative proof scripts and combine the advantages of the procedural and the declarative style of proof. From a proof planning perspective, declarative tactics provide a new format to encode common patterns of reasoning, which is a main issue when trying to automate human problem solving behavior. In particular, declarative tactics provide a major means for global search control in the spirit of proof planning, as they naturally support the refinement of abstract proof plans to formal proofs. The fact that the majority of the knowledge is declaratively encoded within the proof document is a major step to increase the usability of proof planners by users, as well as to provide a possibility to evaluate proof planning systems, as all knowledge becomes explicit. The separation of the strategy language from the programming language plays an essential role in the overall design of our system and provides benefits both for the maintenance of proof planning systems, but also for the runtime behavior, as optimizations can easily be incorporated and local decisions are treated locally.

We evaluated the different aspects of our framework on four case studies with the following results: (i) The class of Statman tautologies, whose proof size is in $O(2^n)$ in the sequent calculus without cut, can be automated efficiently in $O(n^2)$ in a goal directed way within our formalism. (ii) In the domain of set theory, generic assertion level reasoning allows the automation of TPTP benchmark problems which are beyond the scope of most automated theorem provers. Thus, our approach does not only reduce the size of the resulting proofs, but is also beneficial for proof automation. (iii) Based on abstract assertion level reasoning and search space restrictions expressed in the strategy language, a corpus of tutorial dialogues collected in the context of the DIALOG project can be correctly classified. The generated proofs build the basis for further analysis, such as granularity, which is not possible in other approaches. Moreover, incomplete and underspecified steps can be verified. (iv) The so-called limit domain is automated by heuristics. Compared with a previous reference implementation, it is shown that the strategy language drastically reduces the specification costs of proof strategies measured in lines of code, increases the readability and maintainability of the strategies, and allows the formulation of more sophisticated control flows that could not be expressed before.

Overall, the general techniques developed in this thesis contribute both to the community of interactive theorem proving and the community of proof planning. Moreover, the abstract nature of the generated proofs opens new applications for theorem provers where the comprehensibility of proof steps is crucial, such as the computer-assisted teaching of mathematical proofs. The system has successfully been used in the context of automated reasoning and proof tutoring in the ACTIVEMATH system.

Part I

Introduction

1 Introduction

The notion of a mathematical proof has a long tradition and lies at the heart of mathematics. It represents a compelling argument that some statement of interest is necessarily true, as expressed by the equation "Proof = Guarantee + Explanation" (see [Rob00]). Mechanization of mathematical proof describes the computer-supported process of developing mathematical proofs in a format such that it can be checked by a machine, and has been classified as "potentially one of the most interesting and useful applications of automatic computers." (see [McC62]).

Even though efficient automated reasoning procedures have been developed within the last decades (see for instance [Rob65, And81, Bib81, NR01, BGML01]) and even been used to prove open conjectures (see e.g. [McC97]), the mechanization of proof search is usually a complex undertaking. This is not only because the proofs themselves are often very difficult, but also for the following reasons:

- Automated theorem provers are often unable to prove non-trivial theorems due to their time and space requirements in practice, even if they could in theory. Moreover, there are also rich formalisms that cannot be fully automated in principle. As a result, many proofs need to be constructed interactively.

- Formal proofs require the use of a strictly formal language, and the level of detail of a formal proof is significantly higher than in human proofs. To be machine-checkable, all these details need to be filled in (see for example [Geu09, Wie05] for a discussion), increasing the size of the formal proof. This "formal noise" hides the line of reasoning that can be followed and understood by humans, and the intelligible presentation of formal proofs is still under research.

- Automated theorem provers are usually restricted to normal forms. While this results in simple and efficient calculi with only a few inference rules, the normalization process destroys the structure of the formula and introduces copies of subformulas that are distributed over several clauses. This is not an option in the interactive setting, where the user needs to exploit the relationship between the input specification and the proof state to fix the specification in the case that the proof attempt fails. Therefore, one essential requirement is to communicate (partial) proofs in a natural format (see for example [Bun99]). Consequently, most interactive provers are based on a variant of the sequent calculus or natural deduction.

- While being relatively human-readable, sequent calculi are less suited for automation compared to machine oriented calculi, such as resolution. One reason for this can be found in the restriction that inferences may only be applied to top-level formulas and thus must follow the logical connectives to extract subformulas that are needed for the proof. This is not only tedious in the interactive setting[1], but also has the effect that for certain classes of problems the proof size grows exponentially with the size of the formula when no cut is used, such as for the so-called *Statman* tautologies [Sta78]. Indeed, decomposition of formulas can be seen as normalization to clause form, which leads to an explosion of the number of literals [And81]. Nevertheless, an interactive prover should support automation of proofs or proof parts whenever possible.

- To allow proof construction at a more abstract level, several macro facilities, such as *tactics* [GMW79] or *macetes* [FGT92] have been developed. However, tactics were mainly designed to support the interactive discovery of a proof, rather than to represent a proof in a readable format. Indeed, tactic proofs, also called *procedural proofs*, are considered to be difficult to read, and therefore difficult to modify and to maintain (see for example [Zam99] or [Har96c] for a general discussion).

  As an alternative to the procedural proof, the so-called *declarative style* of proof has been developed with the objective of providing a readable format for proofs. This comes at the expense that declarative proofs are more tedious to write. While being readable, the declarative style of proof is not sufficiently automated, and there is no analog to procedural tactics.

- During the development of a theory, it is usually necessary to extend the automation facilities by writing tactics. However, development of tactics is a very difficult task, because tactics cannot be specified within the proof document, but have to be written in the programming language of the underlying theorem prover (such as ML or Lisp). This is because these languages are designed for writing any kind of programs and are not specialized for writing tactics. In addition, the tactic developer must use the system-specific interface provided by the prover.

The contribution of this thesis attempts to make the mechanization of mathematics more natural by reducing the gap between formal and informal proofs. Our approach is centered around proof documents that are human-readable, human-writable, machine-checkable, and support the specifications of proof strategies within the proof document. We approach this goal from two angles:

A1 We develop an abstract calculus that allows the construction of proofs directly at the *assertion level*. The assertion level (see [Hua96]) is a proof style that comes close to proofs as written by humans. The key inference rule is the application of an assertion, which is a generic term to denote axioms, lemmas, hypotheses or other forms of previously established pieces of mathematical knowledge. Due to their abstract nature, assertion level proofs substantially reduce the noise of formal proofs and lead to an understandable presentation. To further reduce the proof size, we combine the assertion level mechanism with the *deep inference* paradigm, which

[1]in particular in the context of side conditions

generalizes the notion of inference rules in the sequent calculus and allows their application at arbitrary depths inside logical expressions, similar to rewriting rules. As a consequence, decomposition steps that are needed in the sequent calculus or natural deduction become superfluous.

A2 We develop a *strategy language* to express generic search strategies as well as common patterns of reasoning. The underlying motivation is to provide generic facilities to automate as many subtasks as possible. The language provides constructs to control the increased non-determinism arising from the deep inference feature and allows its adjustment according to the needs of the application given at hand. Moreover, it explicitly supports the automation of the declarative style of proof and provides fine-grained control over stylistic choices, such as the granularity of the resulting proof scripts, naming of important formulas that are expected to play a major role in the subsequent proof, or the choice between forward and backward steps.

In particular, our approach naturally provides two possibilities to automate the declarative style of proof. First, abstract assertion level proofs can directly be translated to declarative proof scripts. Second, we introduce the notion of a *declarative strategy*, which is defined as an extension of a declarative proof language and directly operates on declarative proof scripts.

1.1 Contributions

We classify the contributions of this thesis with respect to (A1) and (A2). With regard to (A1), the contributions are as follows:

(i) We show that it is possible to directly search for a proof at the assertion level. Compared to the sequent calculus, the resulting proofs are both shorter and involve fewer formulas; they are therefore more readable. Due to their natural form they can directly be presented to the user without the necessity of being translated, allowing for a direct integration into applications. In particular, failed proof attempts can be analyzed in order to detect and fix the errors. Note that while the assertion level has already been proposed two decades ago, so far, proofs were not directly constructed at the assertion level, but had to be obtained by transformation of an underlying proof in natural deduction or resolution. Moreover, only proof parts of a specific form could be abstracted.

(ii) We show that assertion level proof search is complete for first order logic.

(iii) We show that it is possible to combine the assertion level calculus with deep inference. This obviates the decomposition of formulas and allows for exponentially shorter proofs. In particular it preserves the structure of the goal and avoids its normalization via decomposition.

While the resulting calculus is interesting in its own right, we also investigate its practical use:

(iv) We show that the assertion level is not only advantageous for proof presentation, but also for proof automation, based on theorems in the domain of set theory. Set theory is a difficult domain for automated reasoning programs, as it is based on only a few primitives and many axioms. We show that a set of TPTP problems that cannot be proved by most state of the art reasoning systems can be automated efficiently by our approach with generic strategies.

(v) Similarly, we show that deep inference brings practical benefits in terms of runtime. As an example, we show that we can efficiently solve the class of so-called *Statman tautologies*[2] in $O(n^2)$ steps relying only on invertible deep rule applications, which can therefore be seen as some kind of simplification. Thereby, we outperform classical reasoners. We illustrate how this simplification technique can easily be specified within the new developed strategy language.

(vi) We explore the connection between assertion level proofs and declarative proof scripts and show how to translate assertion level proofs to declarative proof scripts. Conversely, we show that underspecified declarative proof scripts can be translated to declarative proof plans, whose gaps can be filled automatically by proof strategies.

The contributions with respect to (A2) are:

(i) Allowing deep inference has the consequence that the search space has a higher branching factor compared to shallow inference, as more inferences are applicable at each step due to the access of subformulas. By just extending an inference system to deep inference, redundancies are introduced to the search space, as new proofs for the same theorem become possible. Moreover, depending on the depth of a rule application, the proof steps might become difficult to understand for a human. For practical applications one therefore has to consider the trade-off between shorter proofs and higher nondeterminism as well as the interplay between the inference rules and the deep inference feature carefully. Therefore, we introduce an inference language to define a calculus and thus giving the user full control over the structure of the search space. The language provides constructs to control the application of the rules. In particular, it provides control constructs to restrict the deep application of the inference rules, covering the full range from full deep inference to top-level inference systems. Thus, classical systems can be obtained as a special case within our framework.

(ii) We provide a strategy language that is independent of the underlying programming language such that it fully supports the document centric approach. The five highlights of the strategy language are: (a) A query language to *dynamically retrieve* relevant knowledge from structured theories, resulting in *adaptive proof strategies.* (b) Control of the deep inference features and context dependent restrictions of tactic modifications. (c) Horizontal as well as vertical refinements[3] of proof sketches being expressed in a declarative language. (d) Dynamic declarative proof scripts as declarative specification language of proof plans. (e) *Efficiency* by *compiling* the control information to executable programs.

Of particular interest is point (c), as it provides new foundational ideas to automate the declarative style of proof. For the community of interactive theorem proving, it provides declarative strategies as an analog to procedural tactics. Declarative strategies are specified within the proof document in a language that the user is

[2]Note that in the sequent calculus without cut the length of the proof is $O(2^n)$

[3]Horizontal refinements describe the modification of a proof sketch at a specific granularity by adding one or several abstract proof steps, whereas vertical refinements describe the refinement of already existing abstract proof steps.

already familiar with. Stylistic choices, such as the granularity of the resulting proof scripts, naming of important formulas that are expected to play a major role in the subsequent proof, or the choice between forward and backward steps, can easily be expressed within the language, which is not possible within the procedural strategies.

From a proof planning perspective, declarative strategies provide a new format to encode common patterns of reasoning, which is a main issue when trying to model human problem solving behavior. In particular, declarative strategies provide a major means for global search control in the spirit of proof planning, as they naturally support the refinement of abstract proof plans to formal proofs. The fact that the majority of the knowledge is declaratively encoded in an intermediate language within the proof document is a major step to increase the usability of proof planners, as well as to provide a possibility to evaluate proof planning systems, as all knowledge becomes explicit. The separation of the strategy language from the programming language plays an essential role in the overall design of our system and provides benefits both for the maintenance of proof planning systems and for the runtime behavior, as optimizations can easily be incorporated and local decisions are treated locally.

These contributions are evaluated as follows:

E1 We show that our approach can successfully be applied to verify underspecified and ambiguous proof steps entered by students in the context of proof tutoring. In 2006, a corpus of tutorial dialogs was collected in the Wizard-of-Oz paradigm between students and experienced mathematics teachers [BHL+06]. The presented techniques allow the correct classification of 95.9%[4] of the corpus, which builds the foundation of further analysis as the constructed proofs are at the assertion level. Due to underspecification this is not possible with standard techniques.

E2 We consider the so-called limit domain (see [Mel98b]) and demonstrate how it can be automated using the developed strategy language. Compared with a reference implementation used by the proof planner Multi [Mei03], we show that the new language drastically reduces the specification costs of proof strategies measured in lines of code, increases the readability and maintainability of the strategies, and allows the formulation of more sophisticated control flows that could not be expressed before.

Overall, the techniques developed within this thesis result in a proof assistant that supports the construction of short and human-readable proofs, i.e., proofs that are machine-checkable but also come with an explanation. This makes it well suited for tasks arising in interactive theorem proving, but, additionally, enlarges the application spectrum of theorem provers to be used for teaching mathematical proof. The system has successfully been used for proof tutoring in the context of the Dialog project [BFG+03] and provided the foundations for a proof step analysis that goes beyond the correctness of proof steps, such as proof granularity [Sch10], which are not featured by other approaches. Within this setting, Schiller showed that a single assertion level step is judged by experts in 90 percent of the cases to be of an appropriate step size.

Publications: Different aspects of the work presented in this thesis have been published in [ABDW08, DB07, SDB07, BDSA07, AD06, DSW08, DS08, DS09, SDB09, DB09, ABDS08, AD10a, BDSA07, AD09, AD10b, ADW10].

[4]The remaining four percent correspond to student input that could not be modeled by the system.

1.2 Outline of the Thesis

This thesis is organized into five parts. The first part, ending with Chapter 2 recapitulates the history and state of the art of interactive and automated theorem proving and its applications to formalizing mathematics and tutoring. The main contributions of the thesis are presented in Part II and Part III.

Part II presents the proof theory underlying the assertion level reasoning. We start with the necessary theoretical background before introducing the notion of an *inference* and defining its application.

Part III presents proof plans as a means to store hierarchical proofs. Subsequently, we develop a specification language for inferences to express *local* search space restrictions. We then present a mechanism to compile such a specification into a program, that uses advantageously the attached control information. Subsequently, we define a strategy language allowing for *global* restrictions of the search space.

Part IV applies the techniques developed in this thesis. First, we show that our framework features the requirements needed in the context of proof tutoring in Chapter 11. In Chapter 12, we show that the assertion level allows the efficient automation of the domain of set theory. Chapter 13 introduces the problem class of Statman tautologies and shows that it can be solved efficiently due to the deep inference mechanism. Finally, we demonstrate how to automate proofs in the context of the so-called limit domain in Chapter 14.

In Part V we summarize the contributions of this work, give a detailed comparison with the previous ΩMEGA system and present an outlook for future research based on the foundations laid in this thesis.

2
Historical Overview and State of the Art

The vision of mechanizing mathematical reasoning dates back at least to Gottfried Wilhelm Leibniz in the 17th century. Leibniz dreamed of a language in which all knowledge could be formally expressed, the “lingua characteristica”, and a calculus of reasoning, the “calculus ratiocinator”, such that two philosophers who disagreed over some problem could say “CALCULEMUS!” to resolve their dispute by calculation.

But it took more than 200 years until the foundations of what today is known as “modern logic” were developed by George Boole and Augustus de Morgan [Boo47, Boo58, Mor47] and others, who developed and studied the calculus of propositional logic. This was extended to first order predicate calculus and Frege’s “Begriffsschrift”[Fre79] separated syntax from semantics and gave one of the first accounts of modern mathematical logic. Subsequently, a variety of different logics were developed and studied, for example by Zermelo [Zer08] Whitehead and Russell [WR10], Fraenkel [Fra22], and Bernays [Ber37]. Higher-order logic in the form of the simply typed λ-calculus was introduced by Church [Chu40]. These new logics were mainly investigated with respect to the development of complete calculi, decidability, and consistency of theories within these logics.

The community suffered a setback after the publication of Gödel’s incompleteness theorems – showing that any consistent theory of a certain expressive strength is incomplete – and the undecidability results of first order logic by Church [Chu36] and Turing [Tur37]. The undecidability result of first order logic was relativized by Herbrand’s theorem [Her30], which essentially allows a certain type of reduction of first-order logic to propositional logic and shows its semi-decidability. A similar result was obtained by Gentzen’s work on cut elimination [Gen69], which can be used to prove a variant of Herbrand’s theorem constructively. Moreover, Gentzen introduced the first human-oriented calculus, natural deduction, aiming at the formalization of a ”natural” way of reasoning. One main feature of natural deduction and related calculi is that they provide a representation of temporary assumptions and hypotheses – a major improvement compared with Hilbert style proofs. The sequent calculus was later reformulated by Beth to obtain the semantic tableau calculus [Bet65].

The first automated theorem provers (ATP) were developed in the late 1950's, like for example the Logic Theorist [NSS57] by Herb Simon and Alan Newell, Gelernter's geometry prover [Gel59], Gilmore's procedure [Gil60], or the Davis Putman method [DP60, DLL62] and the first mechanically generated proof was found by Martin Davis' system. An important step forward was the invention of the resolution calculus and unification by Robinson [Rob65], which is the foundation of many efficient calculi known today.

Since then, a variety of different logics as well as tools to automate them have been developed. These include constructive type theories, temporal logics, process algebras, dynamic logics, and modal logics, to name a just few. However, building theorem provers for these logics is a non-trivial task. Indeed, state of the art theorem provers will often fail to prove non-trivial theorems automatically. As a consequence, *interactive theorem provers* have been developed, where the task is divided between the computer and a mathematician.

2.1 Classical Automated Theorem Proving

We denote by classical automated theorem provers systems that are based upon inference rules especially suited for a computer, for instance because the number of available inference rules is very small. Their strength comes from their ability to traverse a huge search space very efficiently – even though blindly – thereby relying on sophisticated representation techniques to speed up the search and to remove redundancies from the search space. The resulting proofs are at a very low level, hence often very long and not suited for humans, for example, because they rely on non intuitive clausal forms.

For propositional logic, most provers are variants of DPLL [DLL62]. First order or higher-order theorem provers are mainly based on *resolution* [Rob65], *paramodulation* and *superposition* [BGLS92], *tableaux* [Smu68] and the *connection method* [And81, Bib81].

Further developments and refinements resulted in powerful machine oriented theorem provers, some prominent representatives of the field are SATO [Zha97] CHAFF [MMZ$^+$01], and GRASP [SMS96] for propositional logic, BLIKSEM [dN99] MKRP [OS91], OTTER [McC94], VAMPIRE [RV99], SETHEO [Sch97], SPASS [WAB$^+$99], E [Sch02], and WALDMEISTER [HJL99] for first order logic, LEO [BK98, Ben99] and its successor LEO II [BTPF08], SATALLAX [Bro11], and TPS [ASDP90, ABB00] for higher-order logic.

In addition to these general solvers, specialized programs have been designed to prove only particular theorems within a certain class. An example is the four color problem, which states that any map in a plane can be colored using four colors in such a way that regions sharing a common boundary (other than a single point) do not share the same color. The first proof was constructed by Appel and Haken in 1977 [AH77a, AH77b] and heavily relied on a computer program to carry out a gigantic case analysis. Another prominent example of a famous and long standing open conjecture that has been solved automatically is Robbins conjecture, questioning whether Robbins Algebras are Boolean algebras.

2.1.1 Rewriting

Due to the importance of equational reasoning, *term rewriting* has been studied intensely on its own. A good overview of the field can be found in the textbook [BN98] as well as in [DP01]. For the purpose of this thesis, we summarize the most important concepts subsequently. An *equational system* is a set of equations $l = r$. Directed equations $l \to r$ are called *rewrite rules* and are used to replace instances of l by instances of r. A *term* *rewriting system* is a set of rewrite rules. An important concept in rewriting is the concept of a *normal form*. A term is said to be in normal form if it cannot be rewritten further. If each term has a normal form, the term rewriting system is called *terminating*. If the normal form of a term is unique, the rewriting system is called *convergent*. During the rewrite process, it might be the case that several rewrite rules are applicable. If this choice does not matter, the rewrite system is called *confluent*. Thus, convergent is just another description for being confluent and terminating. An important algorithm that tries to transform rewrite system into a confluent one is the *Knuth-Bendix completion* algorithm (see [KB70]). When the algorithm terminates, it can be used to decide whether two terms are equal.

2.2 Interactive Theorem Proving and Proof Style

The AUTOMATH project [dB70, dB73, Bru73, dB94] was the first significant attempt to formalize mathematics and to use a computer to check its correctness. AUTOMATH is a pure proof checker: it does not provide tools for automation, but requires the user to specify each proof step manually. The development of interactive tactic based theorem provers started with the LCF system [GMW79], a system to support automated reasoning in Dana Scott's "Logic for Computable Functions". The main idea was to base the prover on a small trusted kernel, while also allowing for ordinary user extensions without compromising soundness. For that purpose Milner designed the functional programming language ML and embedded LCF to ML. ML allowed the representation of subgoaling strategies by functions, called *tactics*, and to combine them as higher-order functions, called *tacticals*. By declaring an abstract type *theorem* with only simple inference rules, type checking guaranteed that tactics decompose to primitive inference rules, thus ensuring soundness.

Many state of the art interactive theorem provers, such as HOL [AP92], ISABELLE [NPW02], HOL LIGHT [Har96a], COQ [BC04], or NUPRL [CAB$^+$86] are direct descendants of LCF and still based on the same methodology (see for example [BW05, Gor00, Geu09] for an overview and historical remarks). However, they differ in the underlying logic, proof automation, and provide different possibilities to communicate proofs.

An important characteristic which allows the classification of existing systems is the type of *proof language* and *tactic language* provided by the system. With the proof language we understand the language in which the proof is communicated and stored, whereas the tactic language corresponds to the language in which proof schemes, representing common patterns of reasoning, are specified. Note that in the original LCF both proof language and tactic language are realized by the ML language.

2.2.1 Procedural vs. Declarative Proof

State of the art proof assistants provide two main approaches to formalize proofs in a computer: the so-called *procedural style* of proof, and the *declarative style* of proof.

A procedural proof consists of a sequence of tactic applications that reduce the theorem to be proved to trivial facts. While allowing for an efficient execution of recorded proofs, these kind of proofs are difficult to understand. This is because intermediate proof states become only visible when considering the changes caused by the stepwise execution of the tactics. Tactic proofs can be extremely fragile, or reliant on a lot of hidden, assumed details, and are therefore difficult to maintain and modify (see for example [Zam99] or [Har96c] for a general discussion). As the only information during the processing of a proof is the current proof state and the next tactic to be executed, a procedural prover has to stop checking at the first error it encounters.

The ability to define and execute tactics represents a key advancement for interactive theorem provers, as tactics reduce the number of proof commands a user needs to issue to construct a proof. However, tactics do not reduce the length of the proof which is to be checked by a proof checker (or a mathematician), since each tactic has to be expanded to a low level proof which makes use only of calculus level rules. Moreover, proof languages based on tactics were mainly designed to support the interactive discovery of a proof, rather than to represent a proof in a readable format. Indeed, tactic proofs are considered to be difficult to read, and therefore difficult to modify and to maintain. An example of a procedural proof is shown in Figure 2.1(a), which shows a formalization of the proof of the irrationality of $\sqrt{2}$ in the HOL system. The cryptic style is not due to the peculiarity of the HOL system, but typical for a procedural proof due to the facts mentioned above.

An alternative to the procedural proof, a so-called declarative proof, is shown in Figure 2.1(b). In the declarative style, a proof consists of intermediate statements that refer to each other, as shown in Figure 2.1(c).

The roots of the declarative style of proof can be traced back to the AUTOMATH project which addressed the problem of developing a formal language with a natural-language-like syntax that allows for both the exact formalization and for the easy reading and writing of mathematical documents. Whereas the original AUTOMATH language was still quite mechanical, its descendants MATHEMATICAL VERNACULAR [dB94], WEAK TYPE THEORY [KN04], and MATHLANG [KMW04], are close to natural language. Similar to AUTOMATH, the MIZAR system [TB85] pioneered the declarative approach to proof languages. In a declarative proof language, a proof step states *what* is proved at each step, as opposed to a list of interactions required to derive it. They are thus closer to informal mathematics and reasoning and therefore more readable. Moreover, as a declarative proof contains explicit statements for all reasoning steps, a proof checker can recover from errors and continue checking proofs after the first error. It has been noted in [Wen99a] that a proof language can be implemented rather independently of the underlying logic and thus provides an additional abstraction layer. Disadvantages of the declarative approach are that the resulting proofs are longer, therefore more tedious to write, and that they cannot be processed as efficiently as procedural proofs.

Due to its advantage many interactive theorem provers nowadays support declarative proofs (see for example [Sym99, Wen99a, AF06, Cor07, Sym97]). However, the design of a proof language that is both readable and supports the discovery of proof is a non-trivial task. One shortcoming of the declarative approach is that almost all details needed for a formal verification of a proof have to be filled in by the user. Therefore, even declarative, formal proofs still significantly differ from proofs which can be found in mathematical textbooks, because of the standard practice to omit easily inferable proof steps. However, in principle a declarative proof can simply be a sequence of intermediate assertions, acting as islands or step stones between the assumptions and the conclusion (by omitting the constraints indicating how to find a justification of the proof step) leaving the task of closing the gaps to automation tools. Such islands are sometimes also called *proof plans* [DJP06] or *proof sketches* [Wie04]. In recent years, many systems – sometimes called *proof finders* or *proof planner* – have been developed trying to automatically close such gaps, such as MIZAR, NQTHM [BM88], SPL [Zam99], SAD [VLP07], NAPROCHE [KCKS09], SCUNAK [Bro06], TUTCH [ACP01], or the ΩMEGA proof checker [DSW08].

```
local open realTheory transcTheory
in
val SQRT_2_IRRATIONAL = Q.prove (‘ Rational (sqrt 2r)‘, RW_TAC std_ss
[Rational_def,abs, SQRT_POS_LE,REAL_POS] THEN Cases_on ‘q = 0‘ THEN
ASM_REWRITE_TAC [] THEN SPOSE_ NOT_THEN (MP_TAC o Q.AP_TERM ‘\x.  x
pow 2‘) THEN RW_TAC arith_ss [SQRT_POW_2, REAL_POS, REAL_POW_DIV,
REAL_EQ_RDIV_EQ,REAL_LT, REAL_POW_LT] THEN REWRITE_TAC [REAL_OF_NUM_POW,
REAL_MUL, REAL_INJ] THEN PROVE_TAC [lemma])
end;
```

(a) Procedural Proof

```
theorem sqrt 2 is irrational
proof
  assume sqrt 2 is rational;
 then consider i being Integer, n being Nat such that
 W1:  n<>0 and
 W2:  sqrt 2=i/n and
 W3:  for i1 being Integer, n1 being Nat st n1<>0 & sqrt 2=i1/n1 holds
n<=n1 by RAT_1:25;
 A5:  i=sqrt 2*n by W1,XCMPLX_1:88,W2;
 C: sqrt 2>=0 & n>0 by W1,NAT_1:19,SQUARE_1:93;
 then i>=0 by A5,REAL_2:121;
 then reconsider m = i as Nat by INT_1:16;
 A6:  m*m = n*n*(sqrt 2*sqrt 2) by A5
    .= n*n*(sqrt 2)^2 by SQUARE_1:def 3
    .= 2*(n*n) by SQUARE_1:def 4;
 then 2 divides m*m by NAT_1:def 3;
**** remaining 17 lines removed *****
```

(b) Declarative Proof

Assume $\sqrt{2}$ is rational, i.e., there exists natural numbers p, q with no common divisor such that $\sqrt{2} = p/q$. Then $q\sqrt{2} = p$, and thus $2q^2 = p^2$. Hence p^2 is even and, since odd numbers square to odds, p is even; say $p = 2m$. Then $2q^2 = (2m)^2 = 4m^2$, i.e. $q^2 = 2m^2$. But now q^2 is even too and so is q. But then both q and p are even, contradicting the fact that they do not have a common divisor.

(c) Textbook Proof

Figure 2.1: Proof of irrationality of $\sqrt{2}$ in procedural style, declarative style, and in textbook style.

2.2.2 Tactic Languages

Tactics and tactical have first been introduced in the *LCF* system. In [Pau83], Paulson stresses their importance: “The discovery of the operators THEN, ORELSE, and REPEAT, for combining tactics, was a breakthrough in the development of Edinburgh LCF”. Indeed, these operators provide the basic machinery for almost all tactic languages and are available in most interactive theorem provers.

However, tactic development is a very difficult task. This is due to the following reasons: (i) Tactics follow a bottom up approach for proof construction, as they are stepwise built on primitive inference rules and other tactics. Therefore, a tactic developer has to keep in mind the peculiarities of the underlying logic and design the tactic accordingly. (ii) Tactics are usually expressed in the programming language of the underlying theorem prover (such as ML or Lisp), thereby making use of the interface provided by the theorem prover.

The use of the programming language and the use of the interface to the theorem prover often prevent the non-expert user from writing tactics. Even experienced users often prefer a high number of interaction steps to the design of a special purpose proof strategy, even though it could be easily realized in principle. This is because writing a tactic usually forces the user to leave the current proof document, to write the tactic, reload the theory again, and reexecuting the proof to the position where the new tactic can now be applied. This contradicts the *document-centric approach* [ALW06, DF06, GM06, ABDW08], where the main idea is that the document, containing a formal theory and the formal proofs, is the central medium around which tools to assist the author are made available. As a formal theory usually requires the development of several tactics, it is only consequent to integrate these into the document. Also for theoretical purposes the use of a programming language to specify tactics is disadvantageous, because it is very difficult to give tactics a precise semantics.

Moreover, tactic languages and their interfaces vary widely from one theorem prover to the next, with the consequence that tactics cannot be changed between different systems. As a consequence, user seem to stick to a particular theorem prover, as learning and understanding the details of a prover is a difficult task.

A first attempt to separate the tactic layer from the underlying programming language has been realized with COQ’s tactic language $\mathcal{L}_{tac}$ [Del02]. The language is intended to provide a simple to use tactic language layer to bridge the gap between the predefined proof operators and the programming language of the proof assistant. This new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages, which are usually build upon top of procedural languages. However, $\mathcal{L}_{tac}$ remains in the procedural style of the underlying tactic language instead of following the declarative approach of theorem proving. Nevertheless, it already provides pattern matching facilities to match against the current goal state, which can be used to make the structure of the goal state explicit.

In the context of proof planning (see 2.3) Richardson and Smaill present a nondeterministic meta-interpreter (see [Ric02]) which gives a semantic to methodical expressions for the λCLAM system [RSG98]. Similar to tacticals, methodicals combine methods, which are abstractions from tactics and are the planning operators within proof planning [Bun88]. Their meta-interpreter works by stepwise unfolding a continuation, keeping track of the methodical expression to be evaluated. In [Ric02] the language of methodicals is extended and a nondeterministic variant presented, where choice points are pushed on a stack. Note that methodicals are not allowed to backtrack. Similarly, each reasoning state in ISAPLANNER [Dix05], which is a descendant of λCLAM, contains a continuation representing the next reasoning technique to be applied, which expresses in a sense the future of the evaluation. However, methodicals and reasoning techniques are still specified in the underlying programming language.

Several tactic languages exist for rewriting systems. The general idea is to provide a language to specify a class of derivations the user is interested in by controlling the rule applications. For example, most of them provide language constructs to describe preferred application position of rewrite rules, such as bottom-up, top-down, leftmost-innermost or leftmost-outermost. Depending on the language, its constructs are either defined by a combination of low-level primitives or built-in primitives. On a second layer, the languages provide constructs to express choice, sequencing, and recursion. Prominent examples are Aps [Let93], Elan [BK97, BKKR01], Maude [MOMV05], and Stratego [Vis01].

2.2.3 Deduction Modulo, Supernatural Deduction, and Superdeduction

Proofs are usually searched within a context of a specific theory, such as set theory or arithmetic. Therefore, it is crucial to support theory reasoning efficiently. Within interactive theorem proving, the standard approach consists of defining special decision and simplification procedures and to encode them as tactics. However, it is also possible to try to build theory reasoning in the underlying proof theory and to study its properties.

One possibility is to apply the inference rules modulo a congruence $\equiv$ associated with a term rewriting system, which is known as *deduction modulo* [DHK98]. This mechanism is calculus independent and can for example be employed for resolution or natural deduction. Consider for example the standard axiom rule in natural deduction and the correspondent inference rule in deduction modulo, which are shown below, where the equality in the side condition has been replaced by the congruence:

$$\text{Ax}\,\frac{}{\Gamma \vdash \varphi}\,\psi \in \Gamma \wedge \psi = \varphi \qquad\qquad \text{Ax}\,\frac{}{\Gamma \vdash \varphi}\,\psi \in \Gamma \wedge \psi \equiv \varphi \tag{2.1}$$

For deduction rules with several premises, deduction modulo usually allows the use of the equivalence relation $\equiv$ either on the conclusion or a single premise. For example, given the left identity of the neutral element e in group theory in the form of the rewrite rule $e * x = x$, uniqueness of the neutral element can be shown as follows:

$$\cfrac{\cfrac{\cfrac{}{\forall y.y * e' = y \vdash \forall y.y * e' = y}\,\text{Ax}}{\forall y.(y * e' = y) \vdash e * e' = e}\,\forall_E}{\forall y.y * e' = y \Rightarrow e' = e}\,\Rightarrow_I$$

Note the use of the rewrite rule $e * x = x$ in the $\Rightarrow_I$ step. In practice, confluent and terminating term rewrite systems are used to keep the congruence $\equiv$ decidable.

Similarly, rewrite rules that rewrite propositions can be added to the congruence. For example, the equivalence $\forall A, B.A \subset B \Leftrightarrow \forall x.x \in A \Rightarrow x \in B$ might be formulated as the rewrite rule $A \subset B \to \forall x.x \in A \Rightarrow x \in B$. However, it has been pointed out that rewriting with propositions might be confusing (see [Wac05a] for details). This has lead to *supernatural deduction*, where the idea is to replace equivalences on predicate symbols by new introduction and elimination rules. For example, for $\subset$, we obtain the two deduction rules

$$\subset_I \frac{\Gamma, x \in X \vdash x \in Y}{\Gamma \vdash X \subset X}\, x \notin \mathcal{FV}(\Gamma) \qquad \subset_E \frac{\Gamma \vdash X \subset Y \quad \Gamma \vdash t \in X}{\Gamma \vdash t \in Y} \tag{2.2}$$

In [Wac05a], Wack shows soundness and completeness of this calculus and proves cut elimination with an extended notion of a cut: Cuts are the usual cuts; additional cuts are derivations in which a new introduction rule is immediately followed by an elimination rule. Superdeduction (see [BHK07a], [BHK07b]) carries over these ideas to the sequent calculus.

2.2.4 Proof Transformation and Presentation

To be able to explain a machine found proof in a natural style, but also to be able to integrate proofs in machine-oriented calculi in an interactive prover, researchers developed algorithms to transform proofs in machine-oriented calculi into ND proofs [And80, Mil84, Pfe87, Lin89, Wos90, And91, Mei00a]. Based on natural deduction, attempts have been made to present a proof even in natural language.

The χ-proof system [FM88] was one of the first theorem provers designed with a natural language output component. It has been recognized that the proof object used to generate the natural language output plays an essential role for the quality of the generated output: “Since the mechanism for translating a proof tree into a text is so simple[1], much of the challenge in constructing natural text can be transferred to constructing proof trees: to first generate good text, generate good proof terms.” [FM88].

Several other systems have been equipped with modules to generate natural language or pseudo-natural language: Natural Language Explainer [McD83] was devised as a back end for the natural deduction theorem prover THINKER [Pel86, Pel98]. ILF [Da97] uses templates with canned sentence chunks to verbalize proofs. The same is true for the pseudo-natural language presentation components of COQ [TCK95] and the proof system THEOREMA [BJD98].

However, even in the case of human-oriented calculi, such as natural deduction, proofs can become very complex and contain too much information. Huang [Hua96] realized that in human-written proofs, in contrast, an inference step is often described in terms of the application of a definition, axiom, lemma or theorem, which he collectively called *assertions*. More often than not there are constructs above at the so-called *proof level*, such as, e.g., “by analogy”. He developed techniques to detect assertion applications of a specific form and abstracted them to a single step. The assertion level proved to be much better suited for a subsequent verbalization of the proofs than a traditional calculus [Hua94a, HF96].

Still, even at the assertion level, humans often omit *obvious inferences* to further compress a proof. There has been the attempt to capture the notion of *obvious inference* in order to identify (and hide) obvious steps (see for example [Rud87, Dav81] for an early work). Most recently, Schiller has studied the problem in the context of assertion level proofs and presented an approach to automatically learn obvious inferences from users and to adapt a computer generated proof automatically [Sch10].

Another option to reduce the complexity is to structure the proof hierarchically and to present only the outline of the proof at first. Only on demand, a more detailed proof

[1]Which turned out to be otherwise.

of a specific step is shown. Hierarchical proofs have been advocated by several people, such as Lamport [Lam93] in the context of informal proofs. A similar idea is proposed by Back and colleagues for calculational proofs [BGvW96]. In the context of HOL, Grundy and Langbacka [GL97] developed an algorithm to present hierarchical proofs in a browsable format. In the context of the ΩMEGA system, Cheikhrouhou and Sorge developed a hierarchical proof data structure, called $\mathcal{PDS}$ [CS00, ABD+06], which was not only used for proof presentation, but also for proof search. In particular, the $\mathcal{PDS}$ supports the presentation of a proof plan or proof sketch (see the next section for details about proof planning), and refinement operations to close these gaps. The same idea has been picked up by Denney, who developed the notion of hiproof [DPT06]. Most recently, a tactic language for hiproofs has been proposed in [ADL10]. Another possibility for hierarchical proof construction is provided by a method called *window inference* [RS93]. Window inference allows the user to focus on a particular subformula of the proof state, transforming it and thereby making use of its context, as well as opening subwindows, resulting in a hierarchical structure.

2.3 Proof Planning and Proof Refinement

Motivated by the fact that humans perform proof search at a high level and subsequently refine their ideas, Bundy introduced the notion of proof planning [Bun88] as a technique for guiding the search for a proof at an abstract level. Proof planning tries to capture and encode common patterns of reasoning – such as induction – in a in so-called *methods* which are tactics annotated with pre- and postconditions. Originally, the process of proof planning a theorem was divided into two phases: First, a proof is planned at the level of proof methods. If a proof plan – i.e., a sequence of methods that transform the assumptions of the proof to the conjecture – was found, the corresponding tactics are executed, called *expansion*. In the case that the execution of these tactics fails the proof planner has to try an alternative.

Though there is no general problem to directly perform a search at the level of tactics, it can be advantageous to perform the search at the level of methods: (i) The attached tactic need not be executed. Thus using only the specification can speed up the search. (ii) The specification of the tactic can be incomplete or may contain additional constraints or control knowledge to restrict the search, i.e., to provide some form of guidance of proof search. Hence the search space for a proof plan is usually considerably smaller than the one for a calculus level proof [Bun02, MS99b] while loosing completeness. In particular, methods provide a top-down approach to proof development. (iii) Due to the more abstract search at the level of proof plans and the declarativity of methods, proof planning is regarded to be human oriented. Indeed, many methods try to simulate human problem solving behavior – they can thus be seen as an attempt to make problem solving knowledge explicit. Consequently, proof plans are often better to read. (iv) Proof methods provide a basis for the integration and combination of external algorithmic systems such as computer algebra systems. This is because their results can easily be integrated and checked afterwards.

Nowadays, proof planners also interleave the expansion with the proof planning phase and execute tactics such as simplification directly. However, the general idea of providing techniques to structure and organize the search space and to generate proofs where the reasoning patterns are transparent remains. To make allowance for these developments, one simply denotes the output generated by a proof planner as *proof plan*.

There are three main proof planning systems: The OYSTER-CLAM series [BvHHS90, RSG98], ΩMEGA's proof planner MULTI [MM05b], and ISAPLANNER [Dix05], all of which implement proof planning slightly differently, and all of which use slightly varying terminology (see [DJP06] for a comparison).

Proof planning has been successfully applied in a number of first and higher-order settings, including mathematical induction [BvHHS91], hardware verification [CBSB96], higher-order program synthesis [DLS00], nonstandard analysis [MFS02], limit problems [Mel98a, MS99b, Mel98b] and problems in the domain of residue-classes [MPS02].

Over the years, further enhancements took place, aiming at more flexible methods that are no longer restricted to its tactical origins, and the improvement of the global search behavior:

Methodicals: Methodicals represent program constructs such as repeat or if-then-else and allow the combination of several methods to more complex methods, similar to tacticals for tactics. In particular, they provide a facility to specify what to do next, i.e., after successful application of a (basic) method.

Due to these extensions of methods the differences between a tactic and a method became unclear [Den04a, Den04b].

Proof critics: Proof critics or *failure reasoning* (see [Ire92, Ire96, MM05a]) attempt to make use of information from failed proof attempts. They are used (1) to patch a failed proof plan, (2) to suggest a lemma or a generalization, or (3) to introduce a case split. Originally, critics were developed to handle failures of the rippling method in OYSTER-CLAM. They can significantly expand the reasoning power of proof planning systems. More generally, also backtracking can be understood as a critic. The difference between a critic and a method is that a critic can modify (delete and replace) any part of the proof plan, whereas a method performs a local modification of a branch of the proof plan.

Knowledge based proof planning: Knowledge based proof planning [MS99b] focuses on the incorporation of mathematical knowledge into the planning process, as advocated inter alia by Bledsoe [BB77]. One major step is the separation of the heuristic knowledge contained in the preconditions of a method and its explicit representation in the form of *control rules*. Control rules cannot only reason about the current goals and assumptions, but also about the proof planning history and the proof planning context. Thus, the introduction of control rules makes the control flow more flexible and extends the meta-reasoning facilities of proof planning. In some sense they are similar to the control flow which can be expressed by methodicals, even though this correspondence has not been studied theoretically. Another key step is the integration of other tools in the knowledge base to provide justifications for method applications. For example, ΩMEGA is linked to classical theorem provers, computer algebra systems and constraint solvers to support the proof search. The use of external systems has several advantages: External reasoning systems can make use of special data structures to speed up the search; only the final proof has to be translated back to the proof assistant. Even if an external system does not provide a proof that can be transferred back to the system, their use is beneficial in situations where finding is much harder than checking. This is for example the case for factorization or integration in the context of computer algebra systems.

Proof planning with multiple strategies: Proof planning with multiple strategies [MMS08] is based on the notion of a *strategy*. A strategy is an instance of an algorithm that refines or modifies a partial proof plan. Examples for strategies are instances of planning-, instantiation- and backtracking algorithms. In particular, the possibility to invoke a planning algorithm with a subset of the available methods allows for a further restriction of the search space by tailoring strategies for solving a subproblem. Several strategies can cooperate during the proof process, where this cooperation is guided by strategic meta-reasoning.

Proof plans as declarative proof scripts: One of the main features of proof plans is the abstract representation of a proof. Similarly, the main feature of a declarative proof is that it is human-readable. The connection between proof plans and declarative proof scripts has been recognized and exploited by Dixon [Dix05]: He observed that it is possible to present a proof plan as a declarative proof script and designed ISAPLANNER to generate declarative proofs. To represent unjustified steps, Dixon introduced a "gap" command, which can be annotated with a technique to close the gap.

Despite its success proof planning still has problems in practice: (i) It is not trivial to discover common patterns of reasoning or abstract heuristics to guide the proof search. As for all knowledge based systems, there is the risk to put too much specific knowledge in the knowledge base (*over tuning*) such that eventually all examples can be proved, without being general. Though there are some guidelines in [Bun98] what a common pattern of reasoning is or might be, it is in practice more or less infeasible to check whether the used methods obey those principles or to inspect all control rules which were used for the search. (ii) All proof planning systems have not yet retrieved much practical attention. One reason might be that these systems are difficult to use, to setup, and to customize unless you happen to be a developer. Indeed, similar to the case of tactics, proof planning operators are difficult to specify: the user needs to know the underlying programming language of the proof planner and details about the internals of the implementation. Moreover, the success of proof planners is difficult to judge, as the details of the underlying programming language make it difficult to easily perceive the knowledge encoded in the proof planning operators. (iii) Proof planning is not complete, but neither is human proving. (iv) Even though proof planning claims to be human-oriented and natural in the sense that methods encode common patterns of reasoning, it is nearly impossible for a non logician to successfully encode such a pattern, as these methods are heavily influenced by the underlying calculus, as shown in [BMM$^+$01]. Moreover, in addition to abstract methods there are usually many rules representing calculus steps such as the decomposition of connectives. Consequently many proofs constructed by the technique of proof planning are still not very natural compared with standard proofs that can be found in mathematical textbooks.

2.4 Practical Applications of Theorem Proving

Automated and interactive theorem provers are nowadays commonly used by experts mainly to verify hard- and software, but also in mathematics. More recently, they are also used as domain reasoner in the context of proof tutoring, and the application areas are further growing.

One significant shortcoming of theorem provers that hinders their further dissemination is that they are not fully integrated into or accessible from standard tools that are already routinely employed in practice, like, for instance, standard mathematical text-editors. Integrating formal modeling and reasoning with tools that are routinely employed in specific areas is the key step in promoting the use of formal logic based techniques. To make a step towards this direction, there is the recent trend towards *document-centric approaches*. The main idea is that the document, containing a formal theory and formal proofs, is the central medium around which other tools are built to assist the author.

Several attempts have been made in the spirit of the document-centric approach to integrate theorem provers into mathematical practice, e.g., the THEOREMA project [BJK$^+$97, BJD98], which extends the computer algebra program MATHEMATICA by domain specific theorem provers, the symbolic computing system ANALYTICA [BCZ98] which is also built on top of MATHEMATICA, or the COQ [Coq03] community, which recently connected the WYSIWYG-editor TEXMACS to the COQ proof assistant. Similarly ΩMEGA has been integrated into TEXMACS [vdH01], based on the mediator PLATO [MW07].

Even though the declarative style of theorem proving is a step towards the document-centric approach, the overall goal has not yet been fully realized. The implementation of proof search procedures in an interactive prover still requires to write tactics in the underlying programming language and thus to know about the data structures, functions and existing tactics. Thus, only experts can adapt and extend the systems. Nevertheless, theorem provers are nowadays used within practical applications: for verification (by experts), and within tutoring systems.

2.4.1 Verification of Software, Hardware, and Mathematics

Even though interactive theorem proving is still time consuming and tedious, even for experts, interactive theorem provers have grown far beyond very academic examples and are nowadays used for impressive verification projects in mathematics and computer science. This is because they offer two main benefits: (i) confidence of correctness (ii) automatic assistance with routine parts of a proof. Their main application areas are situations in which the cost of error is too high, as witnessed for example for the famous Pentium FDIV bug [Cor94]. The verification of hard- and software is one of the main industrial applications of interactive theorem proving.

For example, J Strother Moore and Matt Kaufmann used ACL2 to prove the correctness of the floating point division operations of the AMD K5 microprocessor [MLK96]. There are many more industrial applications, see e.g. [Har06] for an overview. There has also been the trend to verify large parts of software: Within the Compcert project, an optimizing compiler from a large subset of C to PowerPC assembly code has been verified (see e.g. [Ler09]). Even more impressive, a complete OS kernel has been verified [KAE$^+$10] and verification of protocols has become standard.

Even in the field of mathematics, formalization and verification enjoys a recent boom, motivated by the fact that complex proofs become difficult to check by humans, as was the case for the proof of Kepler's conjecture [Hal06], which is now in the process of being formally verified.

2.4.2 Tutoring Systems for Mathematics

In recent years, a number of software tools have emerged which support tutoring mathematics. As the field is rather broad, we only present a selection of the existing systems. We classify some systems for teaching mathematics (or logic) with respect to three categories: (i) Teaching systems based on computer algebra (ii) logic tutors, and (iii) systems for learning deductions.

Teaching based on Computer Algebra

A number of tutoring systems in mathematics exist which focus on algebraic manipulations and the modeling of math's problems (e.g. algebra symbolization) rather than deductive proof. Examples are the MsLundquist algebra tutor [HK00, HK02], or Aplusix [RBGL07], Buggy [BB78], Slopert [Zin06], AlgeBrain [SRA99], ActiveMath [Mel05], and Beetle [CDF+07].

While these systems are very interesting and valuable when studying general tutoring techniques, such as user modeling or hinting strategies, they do not require sophisticated domain reasoning. Either, the functionality of a computer algebra system is used, or a rather simple rewrite system if a trace of the computation is required for an analysis. For example, derivatives can easily be computed by means of a confluent term rewriting system, such that no search has to be performed. These computations are generally unique and do not have to deal with underspecification and ambiguity.

Logic Tutors

It has been recognized that in addition to computational tasks, students must learn how to conduct mathematical argumentation and mathematical proofs [KAB+04]. Therefore tools have been developed which teach proving skills. For propositional and first order logic examples are the CMU Proof Tutor [SS94], Easy , P Logic Tutor [LLB02], Alfie [vS00], ProofWeb [KWHvR07], Jape [SB96] and Winke [DE98], and for higher-order logic, ETPS [ABP+04]. These systems focus on pure logic and support proof construction using for example Fitch-style diagrams or trees.

Teaching Systems for Mathematical Proofs

There have also been attempts to support teaching mathematics at a more abstract level, comparable to the type of mathematics taught in schools. These include the Epgy Theorem Proving Environment [SN04, MRS01], Mathpert [Bee92], Easy [GBK08], Tutch [ACP01], and Proofnavigator [Sch09], which is a tool for teaching inductive proofs in program verification tasks. In the specialized field of teaching geometric proofs, a variety of systems such as Advanced Geometry Tutor [MV05], Angle [KA93], and Baghera [WBPB01] exist. Note that the fragment of geometry theorem proving is decidable [Har09]. Another example is the Dialog project [BFG+03, BHKK+07a], which had the final goal of natural tutorial dialog between a student and a mathematical assistance system. The ActiveMath [GUM+04] is a learning environment, which provides a fully-fledged learning environment for mathematics.

To teach this kind of mathematics, a tutor system must be able to interact with the student at a level of reasoning similar to that which the student uses to write a proof. The student should be free to express mathematical arguments in a natural way, and not be restricted, for instance, to rule applications in the underlying logic. Therefore, a key point in the design of teaching tools for mathematical proofs is the ability to close the gap between formal proofs and proofs as they are written by students. Filling such gaps automatically seems reasonable, as the problems considered for tutoring are rather trivial from a mathematical point of view and are therefore easier to automate. Staying within the low-level of a logic is not an option as argued by Abel *et al.*: "Larger proofs ... were tedious since each step had to be a single natural deduction inference. For practical reasoning this restriction is highly unsatisfactory..." [ACP01].

2.5 Summary

In this chapter we have surveyed the main developments that build the foundations of computer assisted formal reasoning as it is known today. We have pointed out the difference between proof discovery and presentation, as well as the importance of readability and maintainability of proofs within the interactive setting. Moreover, we presented two proof styles that are supported by current proof assistants: The procedural style which is purely based on tactics and hides intermediate proof states, and the declarative style which is more readable as it makes intermediate proof states explicit. Moreover, we have presented approaches that allow the generation of more abstract proofs automatically, namely proof planning that works above the underlying calculus, and deduction modulo, supernatural deduction and superdeduction which build theory knowledge into the calculus.

Part II

Assertion Level Proofs

3 Assertion Level Proofs

Proofs contain mathematical knowledge and for mathematical knowledge management it is important to represent this adequately. While tactics facilitate interactive proof construction, they do not reduce the length of the proof which is to be checked by a proof checker or by a mathematician, since each tactic has to be expanded to a low level proof which makes use only of calculus level rules. Moreover, though it might be easy to guess in some situations what the result of a tactic application is, “a HOL script looks nothing like a textbook proof. Even HOL experts cannot really read a typical HOL proof without replaying it in a session.” (see [Har96b], p. 2).

Indeed, it has been noted that “a major barrier to more common use of mechanical theorem provers in both software and hardware verification, or verification of mathematical results in general, is the distance between the proof style natural for a mathematician and the proof style supported in various mechanical theorem provers” [AH97]. However, also from a practical point of view there is the need to search for the proofs at a more abstract level, as already noted in [ST89]: “it is practically impossible to prove theorems in calculus, using only basic logical rules”.

To present machine generated proofs at a more abstract level, techniques have been developed to convert (completed) resolution proofs or matrix proofs into human-oriented calculi, (see [And80, Mil84, Pfe87, Lin89, Wos90, And91, Mei00a]). However, as analyzed in [Hua99], although individual steps in natural deduction can be understood easily, the entire proof is still usually at a very low level of abstraction and contains too many steps to be adequate as input for a presentation in natural language.

To come close to the style of proofs as done by humans, Huang [Hua94b, Hua96] introduced the *assertion-level*, where individual proof steps are justified by axioms, definitions, or theorems, or even above at the so-called *proof level*, such as “by analogy”. The idea of the assertion-level is, for instance, that given the facts $U \subset V$ and $V \subset W$ we can prove $U \subset W$ directly using the assertion:

$$\subset_{Trans}\colon \forall U.\forall V.\forall W.U \subset V \wedge V \subset W \Rightarrow U \subset W$$

An assertion level step usually subsumes several deduction steps in a standard calculus, say the classical sequent calculus [Gen69]. Therefore, traditional theorem provers can only achieve such conclusions after a number of proof steps. To use an assertion in the classical sequent calculus, it must be present in the antecedent of the sequent and be processed by means of decomposition rules, usually leading to new branches in the derivation tree. Some of these branches are subsequently closed by means of the axiom rule which correspond to "using" that assertion on known facts or goals.

Huang characterizes assertions as macro steps and describes the following two ways for acquiring new assertion level rules: (i) learning by chunking-and-variablization, (ii) learning by contraposition. The former tries to find and remember repeated applications of an assertion as a new rule by inspecting proofs of a theory that have already be performed. The latter derives a new rule from a given rule by the following rule schema: If r is an existing rule of the form

$$\frac{\Gamma \vdash p_1 \quad \ldots \quad \Gamma \vdash p_n}{\Gamma \vdash q} \tag{3.1}$$

then r' can be acquired by contraposition:

$$r' = \frac{\Gamma \vdash p_1 \quad \ldots \quad \Gamma \vdash p_{i-1} \quad \Gamma \vdash p_{i+1} \quad \ldots \Gamma \vdash p_n \quad \Gamma \vdash \neg q}{\Gamma \vdash \neg p_i} \tag{3.2}$$

For instance, for the inference rule

$$\frac{\Gamma \vdash a \in U \quad \Gamma \vdash U \subset F}{\Gamma \vdash a \in F} \tag{3.3}$$

the following two other rules can be derived by contraposition

$$\frac{\Gamma \vdash a \in U \quad \Gamma \vdash a \notin F}{\Gamma \vdash U \not\subset F} \tag{3.4}$$

$$\frac{\Gamma \vdash a \notin F \quad \Gamma \vdash U \subset F}{\Gamma \vdash a \notin U} \tag{3.5}$$

Huang followed the approach of a human-oriented proof style and developed an algorithm to abstract a given ND proof to the assertion level by finding assertion applications and replacing them by a single rule. His algorithm had to consider each proof step individually and was therefore very expensive from a computational point of view. Moreover, he noted that "the algorithm works well on neatly structured ND proofs, but performs very poorly on machine generated proofs that are mainly indirect" ([Hua99] p. 13). This is because proof steps are often twisted in machine-generated proofs and need to be reordered before a transformation becomes possible. Indeed, it has been noted in [Mei00b] (p. 52) that there is "no practically usable algorithm to abstract 'bad' ND proofs".

Huang was mainly concerned with using the abstract representation at the assertion level for proof presentation in natural language [Hua96, Fie01]; therefore there was no proof theoretic foundation for the assertion level. In particular, it was not possible to directly search for a proof at the assertion level. The reconstruction approach had the disadvantage that only proof parts of a specific form – corresponding to previously extracted assertions – could be abstracted. Moreover, due to the complexity of the abstraction mechanisms the abstraction had to be restricted to global assertions, i.e., definitions and theorems belonging to the theory. Local assertions such as an induction hypothesis were not considered.

It has been argued that assertion level should serve as a basis on which knowledge based proof planning should be based: "We are convinced that it will be possible to overcome at least some of the identified limitations and problems of proof planning as discussed in [BMM$^+$01, Bun02], in particular those that are caused by an unfortunate intertwining of the proof planning and calculus level theorem proving." [VBA03]. However, this has not been studied so far. Therefore, in this thesis we will face this challenge and base the theorem proving process directly on the assertion level. This has the following advantages:

- Assertion level proofs come close to hand written proofs (Contribution A1, Section 1.1). Therefore, they do not only provide a correctness guarantee, but also an explanation. Consequently, they become accessible to a broader audience. In particular, they can naturally be used in the domain of proof tutoring (Evaluation E1, Section 1.1).
- Assertion level proofs can naturally be translated to declarative proof scripts (Contribution A1(vi), Section 1.1), which have become the standard means to communicate proofs in an interactive theorem prover. Therefore, (partial) proofs as well as failed proof attempts can be presented at an abstract level. Note that this is nontrivial, and not supported in state of the art interactive theorem provers, such as Isabelle or Coq: "There is currently no way to transform internal system-level representations of Isabelle proofs back to Isar text" (see [Wen99b], p. 11).
- The use of assertions naturally leads to a goal directed proof search, allowing for efficient proof search procedures that even outperform classical reasoners in specific domains (Contribution A1(iv), Section 1.1). More importantly, it generates much fewer sequents until a proof is found. In many cases, a proof can be found without any search at all by exploiting some abstract reduction properties, which get lost when normalizing the assertion.
- The problem of handling multiplicities of the quantifiers, which is the main problem in first order logic, can be handled for assertions in a natural way, as quantifiers belonging to a single assertion are grouped together.
- The assertion level allows the natural specification of heuristics which cannot easily be formulated at a lower level of abstraction (Contribution A2(i), Section 1.1).

3.1 Examples of Assertion Applications

Before presenting the technical details of the assertion level, we give several examples illustrating the difference between an assertion level justification – including deep assertion applications – and the corresponding justification in the sequent calculus. The key points which are illustrated are:

- Assertion level proofs are shorter than corresponding proofs in sequent calculus or natural deduction.
- Deep inference further shortens the proof size and makes rule application invariant under slight reformulations of the problem.

The first example shows the use of a proper definition at the assertion level, i.e., a formula $\forall \overline{x}.P \Leftrightarrow Q$, where P is a predicate and Q is not necessarily atomic. This simple structure falls in the category which can be handled within superdeduction [BHK07a, BHK07b] and supernatural deduction [BDW07], which follows a similar motivation as the assertion level. It is important to note that the lifting of an assertion to a rule is more than just a "macro" collapsing a sequence of introductions into a single one: The formula corresponding to the assertion is replaced by a rule, while the formula itself is removed from the proof state. This results in non-trivial proof theoretical questions, such as whether the lifting of assertions is a safe operation, i.e., whether provability is preserved when lifting an assertion. From a user's perspective, the resulting sequents get much more readable, as the size of the antecedent is reduced. Moreover, the operational representation of an assertion as inference reflects its effective use as a means to transform a proof state.

Example 3.1.1. *Consider the assertion step that derives $a_1 \in V_1$ from $U_1 \subseteq V_1$ and $a_1 \in U_1$ which we assume to be contained in Γ. The corresponding sequent calculus proof is:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \forall U, V.\, U \subseteq V \Leftrightarrow \forall x.x \in U \Rightarrow x \in V \vdash \Delta}{\Gamma, \forall V.\, U_1 \subseteq V \Leftrightarrow \forall x.x \in U_1 \Rightarrow x \in V \vdash \Delta}\,\forall_L}{\Gamma, U_1 \subseteq V_1 \Leftrightarrow \forall x.x \in U_1 \Rightarrow x \in V_1 \vdash \Delta}\,\forall_L}{\Gamma, U_1 \subseteq V_1 \Rightarrow \forall x.x \in U_1 \Rightarrow x \in V_1 \vdash \Delta}\,\Leftrightarrow_L \quad \dfrac{}{\Gamma \vdash U_1 \subseteq V_1 \Delta}\,\text{Ax}}{\Gamma, \forall x.x \in U_1 \Rightarrow x \in V_1 \vdash \Delta}\,\Rightarrow_L}{\Gamma, a_1 \in U_1 \Rightarrow a_1 \in V_1 \vdash \Delta}\,\forall_L \quad \begin{array}{c}\dfrac{}{\Gamma \vdash a_1 \in U_1, \Delta}\,\text{Ax}\\ \vdots \end{array}}{\Gamma, a_1 \in V_1 \vdash \Delta}\,\Rightarrow_L$$

Lifting the assertion $\forall A, B.A \subset B \Leftrightarrow \forall x.x \in A \Rightarrow x \in B$ to the inference level makes the assertion directly available as a rule and allows for the following one step deduction:

$$\dfrac{\Gamma, a_1 \in U, U \subset V, a_1 \in V_1 \vdash \Delta}{\Gamma, a_1 \in U, U \subset V \vdash \Delta}\,\subset\text{-Def}$$

The second example illustrates the use of an assertion that is not an equivalence. As such, it cannot be handled by superdeduction or supernatural deduction. Moreover, it illustrates the different ways the assertion can be applied.

Example 3.1.2. *Consider the assertion*

$$(\subset_{\mathsf{Trans}}) \quad \forall A, B, C.A \subset B \wedge B \subset C \Rightarrow A \subset C$$

and a corresponding derivation shown below, illustrating how the assertion can be used to derive $\Delta = \{U \subset W\}$ under the assumptions $\Gamma = \{U \subset V, V \subset W\}$:

$$\forall_L\,\dfrac{\forall_L\,\dfrac{\forall_L\,\dfrac{\Rightarrow_L\,\dfrac{\wedge_R\,\dfrac{\text{Ax}\,\dfrac{}{\Gamma \vdash U \subset V, \Delta} \quad \dfrac{}{\Gamma \vdash V \subset W, \Delta}\,\text{Ax}}{\Gamma \vdash U \subset V \wedge V \subset W} \quad \dfrac{}{\Gamma, U \subset W \vdash \Delta}\,\text{Ax}}{\Gamma, U \subset V \wedge V \subset W \Rightarrow U \subset W \vdash \Delta}}{\Gamma, \forall C.U \subset V \wedge V \subset C \Rightarrow U \subset C \vdash \Delta}}{\Gamma, \forall B, C.U \subset B \wedge B \subset C \Rightarrow U \subset C \vdash \Delta}}{\Gamma, \forall A, B, C.A \subset B \wedge B \subset C \Rightarrow A \subset C \vdash \underbrace{U \subset W}_{\Delta}}$$

The crucial steps in this derivation are to use the assertion $(\subset_{\mathsf{Trans}})$ with the instantiation $[U/A, V/B, W/C]$ to show $U \subset W$. As before, the other steps can be understood as unfolding or preparation steps, yielding several branches in the derivation tree, some of *which can be closed using the axiom rule and the available facts* Γ *or goals* Δ*. In contrast, at the assertion level, the same derivation looks as follows:*

$$\frac{}{U \subset V, V \subset W \vdash U \subset W}\subset_{trans}$$

The two examples show that the lifting of an initial assertion can be very beneficial. The next example illustrates that it is also beneficial to lift intermediate formulas, such as the induction hypothesis in an inductive proof. Note that the lifted formula is not closed.

Example 3.1.3. *Consider the proof of the following simple statement about natural numbers* $\forall n, m.n < m \Rightarrow \exists u.n + u = m$ *under the assertions*

$$\begin{array}{rc} Nat: & \forall x.x = 0 \vee \exists y.x = s(y) \\ +_s: & \forall n, m.s(n) + m = s(n + m) \\ <_s: & \forall n, m.n < m \Rightarrow s(n) < s(m) \\ =<_0: & \forall n.n < 0 \Rightarrow \bot \\ =_s: & \forall n, m.s(n) = s(m) \Rightarrow n = m \end{array}$$

giving – among others – rise to the following rules:

$$\textsc{Nat}\ \frac{\Gamma, X = 0 \vdash \Delta \qquad \Gamma, X = s(f_{[\exists y.X=s(y)]}(X) \vdash \Delta}{\Gamma \vdash \Delta} \qquad <_s \frac{\Gamma, N < M \vdash \Delta}{\Gamma, s(N) < s(M) \vdash \Delta}$$

$$=<_0 \frac{}{\Gamma, N < 0 \vdash \Delta}$$

In the first rule, $f_{[\exists y.X=s(y)]}$ *corresponds to the Skolem function that is introduced by using the liberalized* δ^{+^+} *rule [BHS93], which can be understood as follows: Standard Skolemization requires that the Skolem function* f *is new with respect to the whole sequent and takes as arguments all variables that occur free in the sequent. Contrary to that, the liberalized* δ^{+^+} *approach allows the use of the same Skolem function for all formulas that are equal modulo* α*-renaming. For example, the same Skolem function can be used for* $\forall x.P(x)$ *and* $\forall y.P(y)$*. We denote such Skolem-Functions by* $f_{[\forall x.F]}$ *where* $[\forall x.F]$ *denotes the set of all formulas* α*-equal to* $\forall x.F$*. Secondly, the arguments to the Skolem function are only all free variables that actually occur in* $\forall x.F$*.*

During the proof, which is done by induction over n, many steps consists of the application of one of these assertions by decomposing and instantiating it in an appropriate way, thereby yielding several branches in the derivation tree, some of which can be closed using available facts and the axiom rule. Consider, for example, the derivation in Figure 3.1, which shows the assertion derivation of $<_s$ *from which the following assertions can be extracted:*

In the example above, we obtain the following possibilities:

$$< ① \frac{\Gamma, u < v, s(u) < s(v) \vdash \Delta}{\Gamma, u < v \vdash \Delta} \qquad < ② \frac{\Gamma \vdash n < m, \Delta}{\Gamma \vdash s(n) < s(m), \Delta}$$

$$< ①② \frac{}{\Gamma, n < m \vdash s(n) < s(m), \Delta}$$

$$\dfrac{\dfrac{\dfrac{\text{Ax}\,\dfrac{\textcircled{1}}{\Gamma \vdash u < v \vdash \Delta} \qquad \dfrac{\textcircled{2}}{\Gamma, s(u) < s(v) \vdash \Delta}\,\text{Ax}}{\Gamma, u < v \Rightarrow s(u) < s(v), u < v \vdash \Delta}\,\Rightarrow_L}{\Gamma, \forall m.u < m \Rightarrow < m \vdash \Delta}\,\forall_L}{\Gamma, \forall n, m.n < m \Rightarrow s(n) < s(m) \vdash \Delta}\,\forall_L$$

Figure 3.1: Example derivation

For instance, if $\Delta = \{\}$ and $\Gamma = \{u < v\}$, then the axiom rule is no longer applicable in ① which gets a new open sequent. Note that this remains a valid derivation if we add arbitrary formulas Γ' to the antecedent or Δ' to the succedent. We get a variety of these inferences depending on which application of axiom rules are enabled by filling the Γ and Δ; these rules all represent one possible application of the assertion. However, if there is not at least one axiom rule application, then we do not consider this as an application of the assertion (otherwise it would always be applicable); moreover, this derivation is somehow superfluous if none of the subformulas of the assertions is used in the proof. Skolem functions introduced by $\forall_R$-rules are always the same, which results from the use of the δ^{++} rule, where we use the same Skolem function for the same formulas. In the case of derived rules, these are always the subformulas of the assertion which are always the same.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{s(n) < 0, m = 0 \vdash \exists u.s(n) + u = 0}\,<_0}{s(n) < 0, m = 0 \vdash \exists u.s(n) + u = m}\,=_s \qquad \begin{array}{c}\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n < f_1(m), m = s(f_1(m)) \vdash n < f_1(m), n + f_1(f_1(m)) = f_1(m)}\,\text{Ax}}{n < f_1(m), m = s(f_1(m)) \vdash n + U = f_1(m)}\,\text{Hyp1}}{n < f_1(m), m = s(f_1(m)) \vdash s(n + U) = s(f_1(m))}\,=_s}{s(n) < s(f_1(m)), m = s(f_1(m)) \vdash s(n + U) = s(f_1(m))}\,<_s}{s(n) < m, m = s(f_1(m)) \vdash \exists u.s(n) + u = m}\,\exists_R,=^* \\ \vdots \end{array}}{s(n) < m \vdash \exists u.s(n) + u = m}\,\text{Nat}}{\vdash s(n) < m \Rightarrow \exists u.s(n) + u = m}\,\Rightarrow_R}{\vdash \forall m.s(n) < m \Rightarrow \exists u.s(n) + u = m}\,\forall_R}{}$$

Figure 3.2: Induction Step of the example

Figure 3.2 shows the induction step of the example statement where the assertions $\{Nat, +_s, , <_s, =<_0, =_s\}$ and the induction hypothesis (which contains free variables)

$$\forall m'.n < m' \Rightarrow \exists u'.n + u' = m'$$

have been lifted to the level of inference rules. The induction hypothesis gives rise to two *new inferences* Hyp$_1$ *and* Hyp$_2$

$$\textsc{Hyp}_1\ \frac{\Gamma \vdash n < M', n + f_1(M') = M', \Delta}{\Gamma \vdash n + f_1(M') = M', \Delta} \quad \textsc{Hyp}_2\ \frac{\Gamma, n < M', n + f_1(M') \vdash \Delta}{\Gamma, n < M' \vdash \Delta}$$

where f_1 *stands for the Skolem constant* $f_{[\exists y.X=s(y)]}$.

3.2 Deep Application

While being particularly suited for human machine interaction, human oriented calculi, such as the sequent calculus, are less suited for automation and the degree of the automation is rather low compared to machine oriented calculi. One of the reasons is that the inference steps have to follow the logical connectives of a formula, which can introduce high branching just because of the structure of the proof. At the same time, it has been noted in [Har96b] that the use of derived rules in forward direction in interactive theorem proving can be very hard, unless the exact structure of the proof is planned before starting the proof. This is because a rule can only be applied in forward direction if all its premises are available as facts in the current proof state. In spite of many enhancements, such as adding ad-hoc rules for modus ponens and modus tollens [DM94, OS88], imposing regularity conditions [DM94, LMG94], using controlled from of cut [DM94, LMG94], or factoring and merging [MR94], exploiting universal variables [Bec97, BP95], and incorporating features of hyper-resolution [BFN96], tableau and sequent calculi are yet not as efficient as other more machine oriented calculi.

Deep inference formalisms allow the application of inference rules deeply inside formulas, thus compensating for the limitation mentioned above. Intuitively, this can be motivated as follows:

In propositional logic, the sequent proof of $\Gamma \vdash \Delta$ is in the worst case of size $O(2^{|\Gamma\cup\Delta|})$ [Urq98]. If we can reduce $\Gamma \vdash \Delta$ to $\Gamma' \vdash \Delta'$ so that its size decreases (e.g. by the application of the axiom rule deeply inside), if only by 1, we would reduce the potential search space at least by half. Thus, we trade off some polynomial processing for an exponential gain: scanning the formula and looking for occurrences of the same formula can be done in polynomial time.

However, let us stress that even though the ability of finding shorter proofs is very beneficial in the interactive setting, strengthening a calculus as above adds non-determinism, i.e., there are more possibilities to proceed at each expansion step. Thus, there is a trade-off between the advantage of shorter proofs and the disadvantage that these short proofs may be harder to be found automatically, because there are more choice points in the search space. Moreover, deep inference steps may be hard to be understood by a human.

We now give several examples of deep inference applications.

The following example is taken from [BHK07b] in which the authors study the relation of rewriting and equivalences. While the additional rules shorten the proof size considerably, we go one step further and illustrate that the proof collapses to a single step derivation when we allow the application of the assertion deeply inside formulas. Moreover, the deep matching makes the search procedure invariant under some simple variations of the problem.

Example 3.2.1. *Consider the axiom*

$$(\mathsf{Trans})\quad \forall x.\forall z.(x \leq z \Leftrightarrow \exists y.(x \leq y \wedge y \leq z))$$

giving rise to the following two inferences:

$$\textsc{Trans}_L\ \dfrac{\Gamma, x \le y, y \le z \vdash \Delta}{\Gamma, x \le z \vdash \Delta}\ y \text{ NEW} \qquad\qquad \textsc{Trans}_R\ \dfrac{\Gamma \vdash x \le y, \Delta \qquad \Gamma \vdash y \le z, \Delta}{\Gamma \vdash x \le z, \Delta}$$

The following example illustrates the difference between a proof in sequent calculus and the corresponding one in the extended deduction system:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{}{a \le b, b \le c \vdash a \le b, a \le c}\text{Ax} \qquad \dfrac{}{a \le b, b \le c \vdash a \le b, a \le c}\text{Ax}}{a \le b, b \le c \vdash a \le b \wedge b \le c, a \le c}\wedge_R}{a \le b, b \le c \vdash \exists y.(a \le y \wedge y \le c), a \le c}\exists_R \\ \vdots\end{array} \qquad \dfrac{}{a \le c, a \le b, b \le c \vdash a \le c}\text{Ax}}{\exists y.(a < y \wedge y < c) \Rightarrow a \le c, a \le b, b \le c \vdash a \le c}\Rightarrow_L}{a \le c \Leftrightarrow \exists y.(a \le y \wedge y \le c), a \le b, b \le c \vdash a \le c}\Leftrightarrow_L}{\forall z.(a \le z \Leftrightarrow \exists y.(a \le y \wedge y \le z)), a \le b, b \le c \vdash a \le c}\forall_L}{\forall x.\forall z.(x \le z \Leftrightarrow \exists y.(x \le y \wedge y \le z)), a \le b, b \le c \vdash a \le c}\forall_L}{\forall x.\forall z.(x \le z \Leftrightarrow \exists y.(x \le y \wedge y \le z)), a \le b \vdash b \le c \Rightarrow a \le c}\Rightarrow_R}{\forall x.\forall z.(x \le z \Leftrightarrow \exists y.(x \le y \wedge y \le z)) \vdash a \le b \Rightarrow b \le c \Rightarrow a \le c}\Rightarrow_R$$

Using the axiom (Trans) *the proof becomes*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \le b, b \le c \vdash b \le c, a \le c}\text{Ax} \qquad \dfrac{}{a \le b, b \le c \vdash a \le b, a \le c}\text{Ax}}{a \le b, b \le c \vdash a \le c}\textsc{Trans}_R}{a \le b \vdash b \le c \Rightarrow a \le c}\Rightarrow_R}{\vdash a \le b \Rightarrow b \le c \Rightarrow a \le c}\Rightarrow_R$$

Going further and allowing for deep matching, the proof reduces further to a single step:

$$\textsc{Trans}_R\ \dfrac{}{\vdash a \le b \Rightarrow b \le c \Rightarrow a \le c}$$

Interestingly, this is independent of slight changes of the formula, such as

$$\textsc{Trans}_R\ \dfrac{}{\vdash a \le b \wedge b \le c \Rightarrow a \le c} \qquad\qquad \textsc{Trans}_R\ \dfrac{}{\vdash b \le c \wedge a \le b \Rightarrow a \le c}$$

$$\textsc{Trans}_R\ \dfrac{}{\vdash b \le c \Rightarrow a \le b \Rightarrow a \le c}$$

Let us further elaborate on problem reformulations.

Example 3.2.2. *In [Sie09] Sieg discusses the dependence of the proof search on slight structural reformulations of a problem which is given in the supplement of [SDM$^+$65]. The problem is to prove*

$$(\neg (K \Rightarrow A) \vee (K \Rightarrow B)) \tag{3.6}$$

under the assumptions

$$(H \vee \neg (A \wedge K)) \tag{3.7}$$
$$\textit{and}\ \ (H \Rightarrow (\neg A \vee B)) \tag{3.8}$$

The reformulations of the problem consist of replacing one or several occurrences of $(\neg X \vee \Delta)$ *or* $(X \vee \neg\Delta)$ *by* $(X \Rightarrow \Delta)$*, respectively* $(\Delta \Rightarrow X)$*.*

Table 3.1 summarizes the case study: Depending on the problem formulation, the number of search steps needed by prover Apros *to solve the problem ranges from* 9 *to* 277*. This is because proof search in* Apros *is driven by the logical structure of the problem: Replacing the conclusion by*

$$(K \Rightarrow A) \Rightarrow (K \Rightarrow B) \tag{3.9}$$

triggers inversion, i.e., the backward use of introduction rules, which transforms the problem to

$$(3.7), (3.8), (K \Rightarrow A)\,, K \vdash B \tag{3.10}$$

These two assumptions simplify the proof search essentially. In particular, when also rewriting the assumptions (3.7) *and* (3.8)*, they allow the deduction of the intermediate facts* A*,* H *and therefore* B*.*

Problem	Search Steps	Proof Length
original	277	77
1. premise modified	273	87
1.+2. premise modified	149	86
1.+2. premise + conclusion modified	9	12
goal modified	18	29

Table 3.1: Influence of slight reformulations to the search space

By allowing for deep inference, the proof search becomes independent of such a reformulation. To see this, we consider the original problem and the extreme case where all disjunctions have been replaced by implications and show how the problem can be solved by a sequence of axiom rule applications. Within both variants, the same sequence of axiom rule applications results in a proof for the conjecture; in particular its length does not change. We mark the places where the axiom rule is applied by boxes. Each axiom rule application involves both a negative and a positive occurrence of a subformula, where the positive subformula is replaced by true[1]*.*

Formulation 1:

$$\left(H^+ \Rightarrow \left(\neg A^+ \vee B^-\right)^-\right)^-, \left(H^- \vee \neg\left(A^+ \wedge \boxed{K^+}\right)^+\right)^- \vdash \left(\neg\left(K^+ \Rightarrow A^-\right)^- \vee \left(\boxed{K^-} \Rightarrow B^+\right)^+\right)^+ \tag{3.11}$$

$$\left(H^+ \Rightarrow \left(\neg A^+ \vee B^-\right)^-\right)^-, \left(H^- \vee \neg A^+\right)^- \vdash \left(\neg\left(\boxed{K^+} \Rightarrow A^-\right)^- \vee \left(\boxed{K^-} \Rightarrow B^+\right)^+\right)^+ \tag{3.12}$$

[1]As we will see later, this is the case because there is no β-related formula on the path from the negative occurrence to the root of the formula.

$$\left(H^+ \Rightarrow \left(\neg \boxed{A^+} \vee B^-\right)^-\right)^-, \left(H^- \vee \neg A^+\right)^- \vdash \left(\neg \left(\boxed{A^-}\right)^+ \vee \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.13}$$

$$\left(H^+ \Rightarrow B^-\right)^-, \left(H^- \vee \neg \boxed{A^+}\right)^- \vdash \left(\neg \left(\boxed{A^-}\right)^+ \vee \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.14}$$

$$\left(\boxed{H^+} \Rightarrow \left(B^-\right)^-\right)^-, \boxed{H^-} \vdash \left(\neg \left(A^-\right)^- \vee \left(K^- \Rightarrow B^+\right)\right)^+ \tag{3.15}$$

$$\boxed{B^-}, H^- \vdash \left(\neg \left(A^-\right)^- \vee K^- \Rightarrow \boxed{B^+}\right)^+ \tag{3.16}$$

Formulation 2:

$$\left(H^+ \Rightarrow \left(A^+ \Rightarrow B^-\right)^-\right)^-, \left(\left(A^+ \wedge \boxed{K^+}\right)^+ \Rightarrow H^-\right)^- \\ \vdash \left(\left(\boxed{K^+} \Rightarrow A^-\right)^- \Rightarrow \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.17}$$

$$\left(H^+ \Rightarrow \left(A^+ \Rightarrow B^-\right)^-\right)^-, \left(A^+ \Rightarrow H^-\right)^- \vdash \left(\left(\boxed{K^+} \Rightarrow A^-\right)^- \Rightarrow \left(\boxed{K^-} \Rightarrow B^+\right)^+\right)^+ \tag{3.18}$$

$$\left(H^+ \Rightarrow \left(\boxed{A^+} \Rightarrow B^-\right)^-\right)^-, \left(A^+ \Rightarrow H^-\right)^- \vdash \left(\boxed{A^-} \Rightarrow \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.19}$$

$$\left(H^+ \Rightarrow B^-\right)^-, \left(\boxed{A^+} \Rightarrow H^-\right)^- \vdash \left(\boxed{A^-} \Rightarrow \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.20}$$

$$\left(\boxed{H^+} \Rightarrow B^-\right)^-, \boxed{H^-} \vdash \left(A^- \Rightarrow \left(K^- \Rightarrow B^+\right)^+\right)^+ \tag{3.21}$$

$$\boxed{B^-}, H^- \vdash \left(A^- \Rightarrow \left(K^- \Rightarrow \boxed{B^+}\right)^+\right)^+ \tag{3.22}$$

Finally, we consider the so-called *Statman* tautologies [Sta78], which are interesting because their proofs grow exponentially with the size of the formula in the cut-free sequent calculus.

Example 3.2.3. *The nth Statman tautology G_n ($n \geq 1$) is defined as follows:*

$$\begin{aligned} F_k &= \bigwedge_{j=1}^{k} (c_j \vee d_j) && \\ A_1 &= c_1 & B_1 &= d_1 \\ A_{i+1} &= F_i \Rightarrow c_{i+1} & B_{i+1} &= F_i \Rightarrow d_{i+1} \end{aligned}$$

$$G_n = ([[(A_1 \vee B_1) \wedge \ldots] \wedge (A_n \vee B_n)]) \Rightarrow (c_n \vee d_n)$$

$$F_1 = (c_1 \vee d_1) \tag{3.23}$$
$$F_2 = (c_1 \vee d_1) \wedge (c_2 \vee d_2) \tag{3.24}$$
$$F_3 = [(c_1 \vee d_1) \wedge (c_2 \vee d_2)] \wedge (c_3 \vee d_3) \tag{3.25}$$

$$A_1 = c_1 \qquad B_1 = d_1 \tag{3.26}$$
$$A_2 = (c_1 \vee d_1) \Rightarrow c_2 \qquad B_2 = (c_1 \vee d_1) \Rightarrow d_2 \tag{3.27}$$
$$A_3 = (c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow c_3 \qquad B_3 = (c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow d_3 \tag{3.28}$$

Thus we get for $n = 2$

$$\vdash ((c_1 \vee d_1) \wedge (((c_1 \vee d_1) \Rightarrow c_2) \vee ((c_1 \vee d_1) \Rightarrow d_2))) \Rightarrow (c_2 \vee d_2) \tag{3.29}$$

and for $n = 3$ we obtain

$$\begin{aligned}(c_1 \vee d_1) \wedge [(c_1 \vee d_1) \Rightarrow c_2 \vee (c_1 \vee d_1) \Rightarrow d_2] \\ \wedge [[(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow c_3] \vee [(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow d_3]] \\ \Rightarrow (c_3 \vee d_3)\end{aligned} \tag{3.30}$$

Theorem 3.2.4 (Statman [Sta78]). *Every proof of G_n in Gentzen's system without cut has size $O(2^n)$.*

This is because the proofs have 2^n branches, illustrated below for $n = 2$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{(c_1 \vee c_2), c_2 \vdash c_2, d_2}}{(c_1 \vee c_2), c_2 \vdash (c_2 \vee d_2)} \qquad \cfrac{}{(c_1 \vee d_1) \vdash c_1 \vee d_1, c_2 \vee d_2}}{(c_1 \vee d_1), ((c_1 \vee d_1) \Rightarrow c_2) \vdash (c_1 \vee c_2)} \qquad \vdots}{(c_1 \vee d_1), (((c_1 \vee d_1) \Rightarrow c_2) \vee (c_1 \vee d_1) \Rightarrow d_2)) \vdash c_2 \vee d_2}}{(c_1 \vee d_1) \wedge (((c_1 \vee d_1) \Rightarrow c_2) \vee (c_1 \vee d_1) \Rightarrow d_2)) \vdash c_2 \vee d_2}}{\vdash (c_1 \vee d_1) \wedge (((c_1 \vee d_1) \Rightarrow c_2) \vee (c_1 \vee d_1) \Rightarrow d_2)) \Rightarrow c_2 \vee d_2}$$

In contrast, the applicability of the inference rules at any depth, as provided by deep inference, allows us to start the construction of the proof from subformulas, that is, from inside out. As a consequence, the decomposition steps that introduce the branching and give rise to the complexity are no longer needed, as shown below:

$$\vdash \left[\boxed{(c_1^- \vee d_1^-)^-} \wedge \left((\boxed{(c_1^+ \vee d_1^+)^+} \Rightarrow c_2^-)^- \vee ((c_1^+ \vee d_1^+)^+ \Rightarrow d_2^-)^-\right)^-\right] \Rightarrow (c_2^+ \vee d_2^+)^+ \tag{3.31}$$

$$\vdash \left[\boxed{(c_1^- \vee d_1^-)^-} \wedge (c_2^- \vee (\boxed{(c_1^+ \vee d_1^+)^+} \Rightarrow d_2^-))\right]^- \Rightarrow (c_2^+ \vee d_2^+)^+ \tag{3.32}$$

$$\vdash \left[(c_1^- \vee d_1^-)^- \wedge \boxed{(c_2^- \vee d_2^-)^-}\right]^- \Rightarrow \boxed{(c_2^+ \vee d_2^+)^+} \tag{3.33}$$

We will show in Chapter 13 that we can prove these problems in $O(n^2)$ due to deep inference, thus the speedup is exponential. While other calculi, such as the matrix or connection calculus, also allow for an efficient solution of the problem, the interesting feature of our approach is that in each step the proof state is manipulated, and the result of the manipulation can be presented to the user. Therefore, partial proofs can be presented to the user. In contrast, a partial set of connections generally does not contain any information that can be understood by a human.

3.3 Summary

In this chapter we introduced the assertion level which was originally developed by Huang in the context of proof presentation and which will become the basic reasoning layer for this thesis. Thereby, our presentation emphasized the relation of assertion applications with existing approaches such as derived rules and supernatural deduction and illustrated how shorter and more readable proofs can be constructed. We then showed how the length of a proof can further be reduced by allowing the application of inference rules deeply inside formulas. We also showed that deep inference makes proof search more robust against reformulations of a problem.

4 Foundations

We shall now present an overview of the basic notions and methodologies on which our proof theory will be based. Our basic representation language is a simply typed lambda calculus [Chu40], enriched by polymorphic functions. We then introduce the semantics for higher-order logic based on the general models of Henkin [Hen50], taken into account Andrews' corrections [And72]. Recently, Henkin models have been generalized in [BBK04], in which Andrews corrections correspond to the so-called property $\mathfrak{q}$. However, the underlying proof theory remains the same. Finally, we present uniform notation and polarities in the style of Smullyan [Smu68] as a convenient means to present the proof theory in a simple and elegant way.

4.1 Syntax, Semantics and Uniform Notation

4.1.1 Syntax

The simply typed lambda calculus ($\lambda^{\rightarrow}$) is a typed interpretation of the lambda calculus with only one type constructor $\rightarrow$ that builds function types. By introducing types, Church restricted the set of all (untyped) lambda terms to those which can get assigned a unique type. Types were originally introduced as a means to avoid paradoxes such as Russell's Paradox [Rus03]. "Let"-polymorphism, also called Rank-1 (prenex) polymorphism, essentially adds type schemas and allows the definition of polymorphic functions. An example is the function length to compute the length of a list, which can be defined independently of the types of the elements.

Definition 4.1.1 (Types)**.** *Let $\mathcal{T}_{\mathcal{B}}$ be a nonempty, finite set of symbols, whose members will be called* base types*. Moreover, let $\mathcal{T}_{\mathcal{V}}$ be a nonempty, finite set of symbols disjoint from $\mathcal{T}_B$, whose members will be called* type variables*. The set $\mathcal{T}$ of* types *is the smallest set satisfying*

- *if $\tau \in \mathcal{T}_{\mathcal{B}}$ then $\tau \in \mathcal{T}$ (called a* base type*)*
- *if $\tau \in \mathcal{T}_{\mathcal{V}}$ then $\tau \in \mathcal{T}$ (called a* type variable*)*
- *if $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$ then $(\tau_1 \rightarrow \tau_2) \in \mathcal{T}$ (called functional types)*

Notation 4.1.2.
- *If $\tau_1, \ldots, \tau_n \in \mathcal{T}$ then $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ stands for $(\tau_1 \to (\tau_2 \to \ldots \to (\tau_{n-1} \to \tau_n) \ldots))$, that is the functional type constructor $\to$ associates to the right.*
- *If $\tau_0, \tau_1, \ldots, \tau_n \in \mathcal{T}$ then $\tau_1 \times \ldots \times \tau_n \to \tau_0$ stands for $(\tau_1 \to (\ldots \to (\tau_n \to \tau_0)))$*

Informally, we think of $\tau_1 \to \tau_2$ as the set of all functions with domain τ_1 and range τ_2 that fulfill a certain property. In the following we assume a fixed set of base types $\mathcal{T_B}$ with $\iota, o \in \mathcal{T_B}$, where o denotes the type of *truth values* and ι denotes the type of *individuals*. We will omit types if they are clear from the context.

Definition 4.1.3 (Signature). *Let $\mathcal{T}$ be a set of types. A* signature *over $\mathcal{T}$ is a collection of sets $(\Sigma_\tau)_{\tau\in\mathcal{T}}$. The elements of Σ_τ are called constants.*

Given a signature Σ, the well-typed λ-terms are defined as follows:

Definition 4.1.4 (λ-terms). *Let Σ be a signature over $\mathcal{T}$, and for each $\tau \in \mathcal{T}$ let $\mathcal{V}_\tau$ be an infinite set of* variables. *For each $\tau \in \mathcal{T}$, we define the typed λ-terms of type τ, denoted by $\mathcal{T}_{\Sigma,\mathcal{V},\tau}$, as follows:*

(i) if $c \in \Sigma_\tau$ then $c \in \mathcal{T}_{\Sigma,\mathcal{V},\tau}$. We say that c is a constant *of type τ, also denoted by c_τ.*

(ii) if $v \in \mathcal{V}_\tau$ then $v \in \mathcal{T}_{\Sigma,\mathcal{V},\tau}$. We say that v is a variable *of type τ, also denoted by v_τ.*

(iii) if $A_{\tau_1\to\tau_2} \in \mathcal{T}_{\Sigma,\mathcal{V},\tau_1\to\tau_2}$ and $B_{\tau_1} \in \mathcal{T}_{\Sigma,\mathcal{V},\tau_1}$ then $(AB) \in \mathcal{T}_{\Sigma,\mathcal{V},\tau_2}$. We say that (AB) is an application *of type τ_2, denoted by $(AB)_{\tau_2}$.*

(iv) if $A_{\tau_1} \in \mathcal{T}_{\Sigma,\mathcal{V},\tau_1}$ and $x_{\tau_2} \in V_{\tau_2}$ then $\lambda x.\ A \in \mathcal{T}_{\Sigma,\mathcal{V},\tau_2\to\tau_1}$. We say that $\lambda x.\ A$ is an abstraction *of type $\tau_2 \to \tau_1$.*

The set of all λ-terms over Σ is defined as

$$\mathcal{T}_{\Sigma,\mathcal{V}} := \bigcup_{\tau\in\mathcal{T}} \mathcal{T}_{\Sigma,\mathcal{V},\tau}$$

Terms of type o are also called formulas, *denoted by* **wff**.

In this thesis we will consider the higher-order logic $\mathcal{CHOL}$, whose signature is given below:

$\mathcal{CHOL}$ - classical higher-order logic Assume $\mathcal{T}$ is the set of higher-order types over an arbitrary set of base types $\mathcal{T_B}$ and additional base type o.
$\Sigma_{CHOL} := \{True_o, False_o, \neg_{o\to o}, \vee_{o\times o\to o}, \wedge_{o\times o\to o}, \Rightarrow_{o\times o\to o}, \Leftrightarrow_{o\times o\to o}, \forall_{(\tau\to o)\to o}, \exists_{(\tau\to o)\to o}$ and $=_{\tau\times\tau\to o}$ for all $\tau \in \mathcal{T}\}$. The $\mathcal{CHOL}$ terms are the λ-terms over $\Sigma_{\mathcal{CHOL}}$.

Substitutions are finite mappings from variables to terms, and extend homomorphically from variables to terms. Formally, they are defined as follows:

Definition 4.1.5 (Substitution). *Let Σ be a higher-order signature over $\mathcal{T}$ and let $\mathcal{V}$ the set of variables over $\mathcal{T}$. A* substitution *is a type preserving function*[1] $\sigma : \mathcal{V} \to \mathcal{T}_{\Sigma,\mathcal{V}}$. *The* homomorphic extension *of σ is the extension of σ to terms, i.e., the application of σ to terms, defined by*

$$\sigma(t) = \begin{cases} \sigma(t) & \textit{if } t \in \mathcal{V} \textit{ or } t \in \Sigma \\ (\sigma(t_1)\sigma(t_2)) & \textit{if } t = (t_1t_2) \\ \lambda y_\tau.\ \sigma[y/y](t') & \textit{if } t = \lambda y_\tau.\ t' \end{cases}$$

[1]i.e., for all variables $x : \tau$, $\sigma(x)$ has also type τ

Here $\sigma[y/x]$ denotes the function which behaves like σ except for x on which it yields y. The domain *of a substitution σ, denoted by $dom(\sigma)$ is the set of variables x for which $\sigma x \neq x$.*

Notation 4.1.6. *Suppose σ is a substitution having finite domain; say $\{x_1, \ldots, x_n\} = dom(\sigma)$, and for each $i = 1, \ldots, n$ $x_i\sigma = t_i$. Then we write $[t_1/x_1, \ldots, t_n/x_n]$. In particular our notation for the identity substitution is $\{\}$.*

Remark 4.1.7. *A substitution σ is* idempotent*, if its homomorphic extension is idempotent, i.e., $\sigma(\sigma(t)) = \sigma(t)$ $\forall t \in \mathcal{T}_{\Sigma,\mathcal{V}}$*

In the following we introduce the *$\beta\eta$ long normal form*, based on the β and η reduction rules. The $\beta\eta$ normal form is unique up to renaming of bound variables (α renaming).

Definition 4.1.8 (Free Variables)**.** *Let $M \in \mathcal{T}_{\Sigma,\mathcal{V}}$ be a term. The set of free variables of M, denoted by $FV(M)$, is inductively defined as follows:*

(i) $FV(x) = \{x\}$ *for* $x \in \mathcal{V}$

(ii) $FV(c) = \{\}$ *for* $c \in \mathcal{T}_\Sigma$

(iii) $FV(MN) = FV(M) \cup FV(N)$

(iv) $FV(\lambda x.\ A) = FV(A) - \{x\}$

Definition 4.1.9 (λ-conversion)**.** *Let $A, B \in \mathcal{T}_{\Sigma,\mathcal{V}1}$ be λ-terms. We define three rules of λ-conversion:*

(i) $\lambda X_{\tau_1}.\ A_{\tau_2} \rightarrow_\alpha \lambda Y_{\tau_1}.\ A_{\tau_2}[Y/X]$, *provided* $Y \notin FV(A)$ *(*α-conversion*)*

(ii) $(\lambda X_{\tau_1}.\ A_{\tau_2})B_{\tau_1} \rightarrow_\beta A_{\tau_2}[B/X]$ *(*β-conversion*)*

(iii) $(\lambda X_{\tau_1}.\ A_{\tau_1\tau_2}X_{\tau_1}) \rightarrow_\eta A_{\tau_1\tau_2}$ *if* $X \notin FV(A)$ *(*η-conversion*)*

If the conversion rule is used from left to right in (ii) or (iii), we speak of reduction, if it is used from right to left we speak of expansion.

Definition 4.1.10 (*$\beta\eta$ long normal form*)**.** *A term $t \in \mathcal{T}_{\Sigma,\mathcal{V}}$ is in $\beta\eta$ long normal form if it cannot be β-reduced or η-expanded.*

Note that a term can always be converted into $\beta\eta$ long normal form (see [Hue76] for details). From now on, we assume that every substitution is idempotent, as this can always be achieved. Furthermore we assume that all terms are in $\beta\eta$ long normal form. We now describe positions in λ-terms by sequences over natural number. The sequence corresponds to a path to the subterm. The empty sequence is denoted by ϵ.

Definition 4.1.11 (Term Positions)**.** *Let $\mathbb{N}^*$ be the set of words over the non negative integers and let ϵ denote the empty word in $\mathbb{N}^*$. Let Σ be a signature, $\mathcal{V}$ variables over $\mathcal{T}$ and $A \in \mathcal{T}_{\Sigma,\mathcal{V}}$ be a term. The set of* term positions *is inductively defined on the structure of A:*

$$O(A) = \begin{cases} \{\epsilon\} & \textit{A is a constant or a variable} \\ \{\epsilon\} \cup \{0.p | p \in O(t')\} & \textit{A is an abstraction of the form } \lambda x.t' \\ \{\epsilon\} \cup \bigcup_{i=0}^{n} \{i.p | p \in pos(t_i)\} & \textit{A is an application of the form } t_0\ t_1\ \ldots\ t_n \end{cases}$$

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

Let $A \in \mathcal{T}_{\Sigma,\mathcal{V}}$ *be a term and* $\pi \in O(A)$ *a term position. The* subterm at term position π, *denoted by* $[A]_\pi$, *is inductively defined as follows:*

$$[A]_\pi = \begin{cases} A & \text{if } \pi = \epsilon \\ [t_i]_p & \text{if } \pi = i.p \text{ and } A = t_0\, t_1\, \ldots\, t_n \\ [t']_p & \text{if } \pi = 0.p \text{ and } A = \lambda x.\, t' \end{cases}$$

Example 4.1.12. *Consider the formula* $A := \lambda x_\iota.\ s_{\iota\iota} x_\iota$. *Then* $O(A) = \{\epsilon, 0, 00, 01\}$. *The subterms of A can be represented in a treelike structure, where the nodes represent the subterms at position* ϵ, 0, 00, 01 *respectively:*

Definition 4.1.13 (Ordering on Term Positions)**.** *Let* p, q *be term positions. Then* $p \leq q$ *if there exists* $p' \in \mathcal{N}^*$ *with* $pp' = q$.

4.1.2 Type Inference – Algorithm $\mathcal{W}$

In practice, we do not require the user to annotate each term with its corresponding type. Rather, we use the Hindley-Milner type inference algorithm $\mathcal{W}$ as a means to automatically derive the *principal type* (most general type) of a given term t. The algorithm relies on type schemes, which are types that can additionally be universally closed at the outermost level.

Definition 4.1.14 (Type Schema)**.** *The set of* type schemas, *denoted by* $\mathcal{TS}$, *is the smallest set satisfying the following conditions:*

- *If* $t \in \mathcal{T}$, *then* $t \in \mathcal{TS}$
- *Let* $t \in \mathcal{T}$ *be a type and* $tv_1, \ldots, tv_n \in \mathcal{T}_\mathcal{V}$ *be type variables. Then* $\forall tv_1 \ldots \forall tv_n.t \in \mathcal{TS}$

A type schema with bound type variables is called generic *or* polymorph, *otherwise it is called* monomorph.

Let us stress here that all universal quantifiers occur in the beginning of a type scheme, i.e., a type τ cannot contain a type scheme σ.

Definition 4.1.15 (Type Substitution)**.** *A* type substitution *is a finite map* $\mathcal{T}_\mathcal{V} \to \mathcal{T}$.

Type schemas allow the binding of/generalization over type variables in order to allow the instantiation of a let expression with different types within an expression. This makes expression as the following typable[2]:

$$\text{let } id = \lambda x.x \text{ in } pair(id(4), id(true)) \tag{4.1}$$

The type environment contains type schemas for declared constants. This binding induces a partition of the type variables in *free* and *bound* type variables. A type schema can be *instantiated* by substituting types for all bound type variables.

Definition 4.1.16 (Free Type Variables)**.** *Let* τ *be a type schema. The free variables* $\mathcal{FV}$[3] *of* τ *are defined as follows:*

[2]Note that id is used with type $int \to int$ as well as with type $o \to o$

[3]we use the same symbol for the free variables of a term and a type as the argument to which the function is applied to allows for a clear distinction

- $\mathcal{FV}(\alpha) = \alpha$ *where* α *is a type variable*
- $\mathcal{FV}(\tau_1 \to \tau_2) = \mathcal{FV}(\tau_1) \cup \mathcal{FV}(\tau_n)$
- $\mathcal{FV}(\forall \tau_1.\tau) = \mathcal{FV}(\tau) \backslash \tau_1$

In the original setting, the type environment Γ is initially empty and can be extended through "let"-expressions. Within our setting, the constants are defined within a theory, similar to the let constructs in the original framework. Formally, a type environment is defined as follows:

Definition 4.1.17 (Type Environment)**.** *Let* Id *be a set of identifiers. A type environment is a mapping* $Id \to \mathcal{TS}$.

We use $gen(\Gamma, \tau)$ to bind the type variables of τ which are not contained within the context Γ, known as *generalization*. Generalization of a type τ can be understood as closing the expression by quantification over the type variables that are free in τ but not free in the type environment Γ. Moreover, we write unify(τ_1, τ_2) to unify types τ_1 and τ_2. In the case that the unification fails, the term on which the type inference was invoked is not typable and a type error is signaled. Otherwise, the substitution is applied to the type environment Γ. In its original form, the algorithm $\mathcal{W}$ is pure functional; in this case a substitution as well as a counter for new type variables are maintained within the algorithm to guarantee that new type variables are always fresh and to keep track of changes in the type environment. In an imperative implementation, those effects can be modeled via side effects. We describe the standard algorithm, which supports let statements to introduce polymorphism. We use the notation $\Gamma \rhd e : \tau$ to indicate that in the type environment Γ, the expression e has type τ; we write $\Gamma(x)$ to describe the look-up of the variable x in the type environment Γ.

$$\textsc{Id}\ \frac{}{\Gamma \rhd x : [\alpha_i/\tau_i]}\ x : \forall \tau_1 \ldots \tau_n.\tau \in \Gamma;\ \alpha_i\ \textsc{new}$$

$$\textsc{Appl}\ \frac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1'}{\Gamma \rhd e_1\ e_2 : \tau_2\sigma}\ \text{unify}(\tau_1, \tau_1') = \sigma \qquad \textsc{Abs}\ \frac{\Gamma \cup x : \alpha \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \Gamma(x) \to \tau_2}\ \alpha\ \textsc{new}$$

$$\textsc{Let}\ \frac{\Gamma \rhd e_1 : \tau \qquad \Gamma \cup x_1 : gen(\Gamma, \tau) \rhd e : \tau'}{\Gamma \rhd let\ x_1 = e_1\ in\ e : \tau'}$$

The algorithm above can be shown to be *sound*: If, given an environment Γ and a term t, the algorithm $\mathcal{W}$ terminates with the resulting type τ, then we can prove that t has type τ. Moreover, it is *complete*, meaning that for every provable type judgement τ for a term t and an environment Γ, the algorithm terminates with an result τ for t and Γ.

4.1.3 Semantics

This section introduces the model-theoretic semantics of $\mathcal{CHOL}$, which is standard: Following Henkin [Hen50] and Andrews [And72] we will define the notion of a general model, which provides a means for giving mathematical meaning to each well-formed expression.

First we define the notion of a frame.

Definition 4.1.18 (Frame)**.** *A* frame *is a $\mathcal{T}$-indexed family $\{D_\tau\}_{\tau\in\mathcal{T}}$ of nonempty domains, such that $D_o = \{\top, \bot\}$ and $D_{\tau_1\to\tau_2}$ is a collection of functions mapping D_{τ_1} into D_{τ_2}, for $\tau_1, \tau_2 \in \mathcal{T}$. The member of D_o are called* truth values*.*

Assignments assign values to variables:

Definition 4.1.19 (Assignment)**.** *Let $\{D_\tau\}_{\tau\in\mathcal{T}}$ be a frame. An* assignment *is a function $\rho : V \to \cup_{\tau\in\mathcal{T}} D_\tau$ such that for each $x_\tau \in V_\tau$ $\rho(x_\tau) \in D_\tau$.*

For the definition of the function spaces in a frame we use $\Lambda x_\tau e_{\tau'}$ to denote a function from D_τ into $D_{\tau'}$ in order to distinguish it from the syntax.

Definition 4.1.20 (Extensional General Models)**.** *A frame $\{D_\tau\}_{\tau\in\mathcal{T}}$ is an extensional general model in the sense of [And72] if it satisfies the following conditions:*

- (a_0) *For each $\tau \in \mathcal{T}$, $D_{\tau\times\tau\to o}$ contains the identity relation q on $D_{\tau\times\tau\to o}$,*
- (a_1) *$D_{o\to o}$ contains the negation function n such that $n(\top) = \bot$ and $n(\bot) = \top$,*
- (a_2) *$D_{o\to o}$ contains $\Lambda x_o \top$ and $\Lambda x_o x_o$. Also, $D_{o\to o}$ contains the alternation function a such that $a(\top) = \Lambda x_o \top$ and $a(\bot) = \Lambda x_o x_o$,*
- (a_3) *For each $\tau \in \mathcal{T}$, $D_{(\tau\to o)\to o}$ contains a function $\pi_{(\tau\to o)\to o}$ such that for all $g \in D_{\tau\to o}$ $\pi_{(\tau\to o)\to o}(g) = \top$ if $g = \Lambda x_\tau \top$,*
- (b) *For all $\tau, \tau' \in \mathcal{T}$ and all $e \in D_\tau$ the function $\Lambda x_{\tau'} e$ is in $D_{\tau'\to\tau}$,*
- (c) *For all $\tau, \tau' \in \mathcal{T}$ the function $\Lambda x_\tau \Lambda y_{\tau'} x_\tau$ is in $D_{\tau\times\tau'\to\tau}$,*
- (d) *For all $\tau, \tau', \tau'' \in \mathcal{T}$, all $x \in D_{\tau\times\tau'\to\tau''}$ and all $y \in D_{\tau\to\tau'}$ the function $\Lambda z_\tau x(z, y(z))$ is in $D_{\tau\to\tau''}$,*
- (e) *For all $\tau, \tau', \tau'' \in \mathcal{T}$ and all $x \in D_{\tau\times\tau'\to\tau''}$ the function $\Lambda y_{\tau\to\tau'} \Lambda z_\tau x(z, y(z))$ is in $D_{(\tau\to\tau')\times\tau\to\tau''}$,*
- (f) *For all $\tau, \tau', \tau'' \in \mathcal{T}$ the function $\Lambda x_{\tau\times\tau'\to\tau''} \Lambda y_{\tau\to\tau'} \Lambda z_\tau x(z, y(z))$ is in $D_{(\tau\times\tau'\to\tau'')\times(\tau\to\tau')\times\tau\to\tau''}$.*

The interpretation of a λ-term t by an extensional general model $M := \{D_\tau\}_\tau$ and with respect to an assignment ρ is the usual interpretation defined by:

- *$M(o) := D_o = \{\top, \bot\}$,*
- *$M^\rho(True) := \top, M^\rho(False) := \bot$, $M^\rho(\neg) := n$, $M^\rho(=_{\tau\times\tau\to o}) := q \in D_{\tau\times\tau\to o}$, and the logical functions $\wedge, \vee, \Rightarrow$, and $\Leftrightarrow$ have the classical interpretation,*
- *$M^\rho(\forall_{(\tau\to o)\to o}) := \pi \in D_{(\tau\to o)\to o}$, and $M^\rho(\exists_{(\tau\to o)\to o}) := M^\rho(\lambda x_{\tau\to o} \neg(\forall(\lambda y_\tau \neg(xy))))$,*
- *$M^\rho(c_\tau) \in D_\tau$, for any constant c_τ,*
- *$M^\rho(x_\tau) := \rho(x_\tau) \in D_\tau$, for any variable x_τ,*
- *$M^\rho(t_0\, t_1, \ldots, t_n) := M^\rho(t_0)(M^\rho(t_1), \ldots, M^\rho(t_n))$,*
- *$M^\rho(\lambda x_\tau t_{\tau'})$ is the function from D_τ to $D_{\tau'}$ that maps every element $e \in D_\tau$ to $M^{\rho[e/x]}(t)$.*

Definition 4.1.21 (Satisfiable, Valid)**.** *A formula Φ is* satisfiable *if there is a model M such that for all variable assignments ρ it holds that $M^{\rho}(\Phi) = \top$. A formula Φ is* valid *if it is satisfiable in all models.*

Notation 4.1.22. *We write $M^{\rho}(\Phi) \models \Phi$ to indicate that model M satisfies Φ. We write $\models \Phi$ to indicate that Φ is valid.*

4.1.4 Uniform notation and Polarities

In this section, we introduce signed formulas, which are simply formulas annotated with a polarity which is either positive ($+$), negative ($-$), or undefined ($\circ$). Intuitively, the polarities encode the side in the sequent calculus on which a subformula will occur when decomposing the formula: If a subformula has negative polarity, it will occur in the antecedent of the sequent, if it has positive polarity, it will occur in the succedent of the sequent.

Definition 4.1.23 (Signed Formula)**.** *Let Σ be a signature over types $\mathcal{T}$. A signed formula is a pair $\langle A, p\rangle$ where $A \in \mathcal{T}_{\Sigma,V,o}$ is a formula and $p \in \{+, -, \circ\}$ is the polarity of A. We write A^p for $\langle A, p\rangle$.*

As we will mainly consider signed formulas, we extend the notion of satisfiability to signed formulas:

Definition 4.1.24 (Satisfiability of Signed Formulas)**.** *Let F^p be a signed formula of polarity p, M a model and ρ an assignment. Then we define $M^{\rho} \models F^+$ if $M^{\rho} \not\models F$. Conversely, $M^{\rho} \models F^-$ if $M^{\rho} \models F$. We extend the notion to sets of formulas $\mathcal{F}$ by $M^{\rho} \models \mathcal{F}$ if $\forall F \in \mathcal{F}.\ M^{\rho} \models F^p$*

The uniform notation assigns *uniform types* to signed formulas. Intuitively, the uniform type encodes the effect of decomposing a formula into its major subformulas in a sequent calculus proof. There are six uniform types: α and β for propositional connectives, γ and δ for quantification over object variables, and ϵ and ζ for equations and equivalences.

The uniform types α, β encode the behavior of decomposing a propositional formula. If a connective has uniform type α and is decomposed using the decomposition rules of the sequent calculus SK, the resulting subformulas occur in the same sequent of the proof. If a connective has uniform type β and is decomposed using the decomposition rules of SK, the proof splits into two branches and both subformulas occur in the different branches.

The uniform notation reduces the number of cases in the definition of a Hintikka set, which we will need later to show the completeness of assertion level proofs, as well as in some other proofs. Moreover, it will be used to statically determine the context of a subformula, based on which transformation rules can be derived.

Example 4.1.25. *The following example illustrates how uniform types and polarities define the behavior of subformulas during the proof. We use subscripts to differentiate between the two occurrences of A, B and C in the sequent.*

$$\begin{prooftree}
\AxiomC{}
\LeftLabel{AXIOM}
\UnaryInfC{$A_1, B_1 \vdash A_2$}
\AxiomC{}
\RightLabel{AXIOM}
\UnaryInfC{$A_1, B_1 \vdash B_2$}
\RightLabel{$\wedge_R$}
\BinaryInfC{$A_1, B_1 \vdash A_2 \wedge B_2$}
\AxiomC{}
\RightLabel{AXIOM}
\UnaryInfC{$A_1, B_1, C_1 \vdash C_2$}
\RightLabel{$\Rightarrow_L$}
\BinaryInfC{$A_1, B_1, (A_2 \wedge B_2 \Rightarrow C_1) \vdash C_2$}
\RightLabel{$\wedge_L$}
\UnaryInfC{$(A_1 \wedge B_1), (A_2 \wedge B_2 \Rightarrow C_1) \vdash C_2$}
\RightLabel{$\wedge_L$}
\UnaryInfC{$(A_1 \wedge B_1) \wedge (A_2 \wedge B_2 \Rightarrow C_1) \vdash C_2$}
\RightLabel{$\Rightarrow_R$}
\UnaryInfC{$\vdash (A_1 \wedge B_1) \wedge (A_2 \wedge B_2 \Rightarrow C_1) \Rightarrow C_2$}
\end{prooftree}$$

If we annotate the formula $(A_1 \wedge B_1) \wedge (A_2 \wedge B_2 \Rightarrow C_1) \Rightarrow C_2$ with polarities and uniform types, we get the following:

$$\left(\left[(A_1^- \wedge^\alpha B_1^-)^- \wedge^\alpha ([A_2^+ \wedge^\beta B_2^+]^+ \Rightarrow^\beta C_1^-)^-\right]^- \Rightarrow^\alpha C_2^+\right)^+ \tag{4.2}$$

Indeed, the single formula C_1 occurs only on the left-hand side on top-level of sequences and the single formula C_2 occurs on top-level on the right-hand side of sequences.

Definition 4.1.26 (Uniform Notation). *(i) A signed formula is of type α if the major subformulas obtained by application of the corresponding decomposition sequent calculus rule both occur in the same sequent.*

(ii) A signed formula is of type β if the application of the corresponding decomposition sequent calculus rule splits the proof, such that the major subformulas occur in different sequences.

(iii) A signed formula is of type γ if the corresponding sequent calculus quantifier elimination rule doesn't impose any restrictions on the instantiations

(iv) A signed formula is of type δ if the corresponding sequent calculus quantifier elimination rule imposes an Eigenvariable condition *, i.e., the inserted variable has to be new.*

The groups and and notions of instances are given in Figure 4.1.

Notation 4.1.27.

- *$\alpha^p(\alpha_1^{p_1}, \alpha_2^{p_2})$ denotes the signed formula of polarity p, uniform type α and subformulas α_i with polarities p_i respectively.*
- *$\beta^p(\beta_1^{p_1}, \beta_2^{p_2})$ denotes the signed formula of polarity p, uniform type β and subformulas β_i with polarities p_i respectively.*
- *$\overline{\beta}^p(\varphi_1^{p_1}, \dots \varphi_n^{p_n})$ denotes a signed formula of polarity p, with subformulas φ_i of polarity p_i, all of which are β-related to each other. It is undefined for the case that $n = 1$ and $p_1 \neq p$.*

$$\overline{\beta}^p(\varphi_1^{p_1}, \dots \varphi_n^{p_n}) := \begin{cases} false^- & \text{if } n = 0 \text{ and } p = - \\ true^+ & \text{if } n = 0 \text{ and } p = + \\ \varphi_1^{p_1} & \text{if } n = 1 \text{ and } p_1 = p \\ \perp & \text{if } n = 1 \text{ and } p_1 \neq p \\ \beta^p(\varphi_1^{p_1}, \varphi_2^{p_2}) & \text{if } n = 2 \\ \beta^p(\varphi_1^{p_1}, \overline{\beta}^p(\varphi_2^{p_2}, \dots, \varphi_n^{p_n})) & \text{else} \end{cases}$$

α	α_1	α_2
$(\Phi \vee \Psi)^+$	Φ^+	Ψ^+
$(\Phi \rightarrow \Psi)^+$	Φ^-	Ψ^+
$(\Phi \wedge \Psi)^-$	Φ^-	Ψ^-
$(\neg\Phi)^+$	Φ^-	
$(\neg\Phi)^-$	Φ^+	

β	β_1	β_2
$(\Phi \wedge \Psi)^+$	Φ^+	Ψ^+
$(\Phi \vee \Psi)^-$	Φ^-	Ψ^-
$(\Phi \Rightarrow \Psi)^-$	Φ^+	Ψ^-

δ	δ_0
$(\forall x\Phi[x])^+$	$\Phi[x/c]^+$
$(\exists x.\Phi[x])^-$	$\Phi[x/c]^-$

γ	γ_0
$(\forall x.\Phi[x])^-$	$\Phi[x/c]^-$
$(\exists x.\Phi[x])^+$	$\Phi[x/c]^+$

ϵ	ϵ_1	ϵ_2
$(\Phi \Leftrightarrow \Psi)^-$	Φ^0	Ψ^0
$(\Phi = \Psi)^-$	Φ^0	Ψ^0

ζ	ζ_0	ζ_1
$(s \Leftrightarrow t)^+$	s^o	t^o
$(s = t)^+$	s^o	t^0

Figure 4.1: uniform notation

Given a formula F and a subformula G of F, we can intuitively think of the α-related subformulas of G in F as formulas which may be used to prove F, i.e., the context of F. We can think of the β-related formulas as conditions which have to be shown to get access to the subformula.

We will use the function $\overline{\beta}$ to construct new formulas containing the conditions of the subformula G.

Definition 4.1.28 (α,β-related). *Let $\Phi(F,G)^p \in \mathcal{T}_{\Sigma,\mathbb{V},o}$ be a signed formula of polarity p with subformulas F and G. We say that F and G are α-related (β-related resp.) in $\Phi(F,G)^p$, if and only if the smallest signed subformula of $\Phi(F,G)$ which contains both F and G is of uniform type α (β resp.).*

Moreover, F and G are strictly α (β-related) to each other, if all nodes on the path between F and G are of type α (β).

Notation 4.1.29. *Let $\Phi[F,G]^p$ be a signed formula with subformulas F and G. If F and G are α-related, we write $F \sim_\alpha G$. Similarly, we write $F \sim_\beta G$, if F and G are β-related.*

Example 4.1.30. *Consider the signed formula $F = (\Phi \Rightarrow \Psi \wedge \Psi')^+$. The smallest signed subformula of F containing both Φ and Ψ' is the whole signed formula F. As this formula is of type α, Φ and Ψ' are α-related.*

Lemma 4.1.31 (Satisfiability of Signed Formulas). *Let M be a model, μ be a variable assignment, φ^p a signed $\mathcal{L}$-formula of polarity p. Then it holds that*

- $M^\mu \models (\alpha(\alpha_1,\alpha_2))$ *iff* $M^\mu \models (\alpha_1)$ *and* $M^\mu \models (\alpha_2)$
- $M^\mu \models (\beta(\beta_1,\beta_2))$ *iff* $M^\mu \models (\beta_1)$ *or* $M^\mu \models (\beta_2)$
- $M^\mu \models \gamma(\varphi)$ *iff* $M^\mu \models (\gamma(t))$ *for all* $t \in U$
- $M^\mu \models (\delta(\varphi)$ *iff* $M^\mu \models (\delta(t))$ *for some* $t \in U$

Proof. □

Definition 4.1.32 (Literals in Signed Formulas). *Let $\varphi(F)^p$ be a signed formula of polarity p. We say that F is a* literal *in $\varphi(F)^p$ if F has a polarity, but none of its subterms.*

4.2 Higher-Order Unification

Unification is the process to find a substitution σ that makes two given input terms s and t syntactically equal, i.e., $s\sigma = t\sigma$. If such a substitution exists, the substitution is called a unifier of s and t. While first-order unification is decidable and all unifiable terms have a unique most general unifier, unification in the higher-order context is much more difficult. Already the second order case has been shown to be undecidable by Goldfarb [Gol81]. In the even more general higher-order case, unification is undecidable and lacks most general unifiers (see [BS94] for a survey). In the sequel, we present a version of the transformation system PT for higher-order preunification of Snyder and Gallier [SG89], which works sufficiently well in practice. As usual, we assume all terms to be in $\beta\eta$ long form, that is, of the form

$$\lambda x_1 \dots x_n . f(u_1 \dots u_p) \tag{4.3}$$

We use the standard terminology and call f the head of the term. If the head of a term is a constant or one of the variables $x_1, \dots, x_n$, it is called *rigid*, otherwise it is called *flexible*. Moreover, we denote a unification problem between s and t by $s =^? t$, and make use of the convenient notation to denote a list of syntactic objects $s_1, \dots, s_n$ where $n \geq 0$ by $\overline{s_n}$. n-fold abstraction and application is written as $\lambda\overline{x_n}.s$ and $a(\overline{s_n})$, representing the terms $\lambda x_1 \dots \lambda x_n.s$ and $(\dots(a\ s_1)\dots s_n)$, respectively.

Preunification differs from unification by the handling of so-called flex-flex pairs, which are treated as constraints and not attempted to be solved. The important insight is that flex-flex pairs

$$\lambda\overline{x}.P(\dots) = \lambda\overline{x}.P'(\dots) \tag{4.4}$$

are guaranteed to have at least one unifier of the form

$$\{P \mapsto \lambda\overline{x_m}.a, P' \mapsto \lambda\overline{x_n}.a\} \tag{4.5}$$

This shrinks the search space, as flex-flex pairs might have an infinite number of unifiers. Rigid-rigid pairs are solved as in the first order case. What remains are flex-rigid pairs. There are two possible strategies, *projection* and *imitation*. In the case of projection, the head of the rigid term occurs within the bound variables. For example, the flex-rigid pair $F(a) =^? a$ can be solved using the projection $F \mapsto \lambda x.x$. In the second case, the idea is to partially substitute for the flexible variable the function of the rigid term. For example, the flex-rigid pair $F(a) =^? a$ can be solved using imitation $F \mapsto \lambda x.a$.

Deletion:

$$\{t =^? t\} \cup S \Rightarrow S \tag{4.6}$$

Decomposition:

$$\{\lambda\overline{x_k}.v(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_n})\} \cup S \Rightarrow \overline{\lambda\overline{x_k}.t_n =^? \lambda\overline{x_k}.t'_n} \cup S \tag{4.7}$$

Elimination:

$$\{F =^? t\} \cup S \Rightarrow \Theta S \tag{4.8}$$

provided that $F \notin \mathcal{FV}(t)$ and $\Theta = \{F \mapsto t\}$

Projection:

$$\{\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(t'_m)\} \cup S \Rightarrow \{\lambda\overline{x_k}.\Theta t_i(\overline{H_j(\overline{t_n})}) =^? \lambda\overline{x_k}.v(\overline{\Theta t'_m})\} \cup \Theta S \tag{4.9}$$

where $\Theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_j(\overline{x_n})})\}$ and H_i new

Imitation:

$$\{\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.f(\overline{t'_m})\} \cup S \Rightarrow \{\overline{\lambda\overline{x_k}.H_m(\overline{\Theta t_n}) =^? \lambda\overline{x_k}.\Theta t'_m}\} \cup \Theta S \tag{4.10}$$

where $\Theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$ and H_i new

Example 4.2.1. *The following example has an infinite number of preunifiers. We show only a single derivation:*

$$\{F(f(a)) =^? f(F(a))\} \tag{4.11}$$

$$\Rightarrow^{Imit} f(G(f(a))) =^? f(f(G(a))); F \mapsto \lambda x.f(G(x)) \tag{4.12}$$

$$\Rightarrow^{Dec} G(f(a)) =^? f(G(a)); F \mapsto \lambda x.f(G(x)) \tag{4.13}$$

$$\Rightarrow^{Proj} f(a) =^? f(a); F \mapsto \lambda x.f(G(x)), G \mapsto \lambda x.x \tag{4.14}$$

4.3 Summary

In this chapter we have formally introduced the syntax and semantics of classical higher-order logic which we will use as foundation within this thesis. We have also presented the typing and unification algorithms that are used within the underlying implementation. We also introduced the uniform notation, which will later be used to characterize the deep application of inference rules.

5 CORE Proof Theory

We shall present the CORE calculus that has been developed by Autexier in his PhD thesis ([Aut03]) and which we will use as underlying calculus. CORE's proof theory is a meta proof theory and supports a variety of logics. Even though the subsequent results can easily be generalized to all logics supported by the CORE framework, we restrict our self to the case of higher-order logic, as it alleviates the subsequent presentation considerably.

The CORE calculus relies on an extension of *extensional expansion proofs* and is based on the idea to construct a proof by transforming parts of a given formula until the trivial formula true is obtained. The formula represents the conjecture to be proven together with the underlying background theory in which the proof attempt takes place. Extensional expansion proofs are extensions of *extensional expansion trees*(see [Pfe87] for details) and support equivalences, equations, the atoms *true* and *false*, and overcome the restriction that the scope of a negation is just one literal. Moreover, they include a rule to dynamically increase the multiplicities of meta-variables on the fly.

A main feature of the CORE calculus is that the logical context of any part of a formula can be statically determined by its structure, formalized in the notion of *replacement rules*. As replacement rules can be applied to any subformula, proof transformation is possible without decomposing the formula, as is the case in the sequent calculus.

Internally, a CORE proof state consists of two complementary parts: an *indexed formula tree* for the initial conjecture (including the theory), and a free variable copy of the indexed formula tree, called *free variable indexed formula tree*. The former is used to check the admissibility of substitutions, while the latter is actively transformed by the CORE calculus rules. Before presenting the calculus in detail, let us state the main result of the calculus:

Theorem 5.0.1 (Soundness and Completeness of the CORE [Aut03])**.** *The* CORE *calculus is sound and complete.*

5.1 Indexed Formula Trees

Following [Aut03], we define indexed formula trees in two steps: First, we define the so-called *initial indexed formula tree*, which is the formula tree initially obtained from a given formula. In a second step, we then add nodes that represent the introduction of Leibniz' equality, extensionality introduction as well as the introduction of cut.

Definition 5.1.1 (Initial Indexed Formula Tree)**.** *We define* initial indexed formula trees *inductively over the structure of formulas. Each node of the tree has a formula as label, a polarity, and a uniform type. All nodes, except for the root node, have also secondary uniform types, which is the uniform type of their parent nodes.*

1. *If A^p is a signed atom of polarity p and without uniform type, then $Q = A^p_{\cdot}$ is an initial indexed formula tree of polarity p and no uniform type, which is indicated by the subscript $\cdot$. Those literal nodes are leaves of indexed formula trees and $\boldsymbol{Label}(Q) := A$.*

2. *If $\epsilon(s,t)^p$ is a signed formula of polarity p and uniform type ϵ, then $Q = \epsilon(s,t)^p_\epsilon$ is an initial indexed formula tree of polarity p and uniform type ϵ. They are also leaves of indexed formula trees and $\boldsymbol{Label}(Q) := \epsilon(s,t)$.*

3. *Let Q' be an initial indexed formula tree of polarity p and $\boldsymbol{Label}(Q') = \varphi$ and $\alpha^{-p}(\varphi^p)$ a signed formula with the opposite polarity $-p$. Then*

$$Q = \begin{tikzcd} \alpha(\varphi)^{-p}_\alpha \arrow[d, dash] \\ Q' \end{tikzcd}$$

is an initial indexed formula tree with $\boldsymbol{Label}(Q) := \alpha(\varphi)$, of polarity $-p$ and uniform type α. The secondary type of Q' is α_1.

4. *Let Q_1, Q_2 be initial indexed formula trees with respective polarities p_1 and p_2, and assume a signed formula $\alpha^p(\boldsymbol{Label}(Q_1)^{p_1}, \boldsymbol{Label}(Q_2)^{p_2})$ of polarity p. Then*

$$Q = \begin{tikzcd} & \alpha(\boldsymbol{Label}(Q_1), \boldsymbol{Label}(Q_2))^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_1 & & Q_2 \end{tikzcd}$$

is an initial indexed formula tree with $\boldsymbol{Label}(Q) := \alpha(\boldsymbol{Label}(Q_1), \boldsymbol{Label}(Q_2)$, polarity p, and uniform type α. The secondary types of Q_1 and Q_2 are α_1 and α_2.

5. *Let Q_1, Q_2 be initial indexed formula trees with respective polarities p_1 and p_2, and assume a signed formula $\beta^p(\boldsymbol{Label}(Q_1)^{p_1}, \boldsymbol{Label}(Q_2)^{p_2})$ of polarity p. Then*

$$Q = \begin{tikzcd} & \beta(\boldsymbol{Label}(Q_1), \boldsymbol{Label}(Q_2))^p_\beta \arrow[dl, dash] \arrow[dr, dash] & \\ Q_1 & & Q_2 \end{tikzcd}$$

is an initial indexed formula tree with $\boldsymbol{Label}(Q) := \beta(\boldsymbol{Label}(Q_1), \boldsymbol{Label}(Q_2)$, of polarity p and uniform type β. The secondary types of Q_1 and Q_2 are β_1 and β_2.

6. *Let $\gamma^p x\varphi(x)$ be a signed formula of polarity p, and $Q_i, 1 \leq i \leq n$ initial indexed formula trees with $\boldsymbol{Label}(Q_i) = \varphi(X_i)$ where the X_i are new (meta) variables. Then*

$$Q = \begin{tikzcd} & \gamma^p x\varphi(x)^p_\gamma \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ Q_1 & \ldots & Q_n \end{tikzcd}$$

is an initial indexed formula tree with ***Label***$(Q) := \gamma^p x\varphi(x)$*, of polarity* p *and uniform type* γ*. All the* Q_i *then have secondary type* γ_0*. The multiplicity of* Q *is* n*. For each* $1 \leq i \leq n$ *we say that* Q_i *is the binding node for* X_i*. We also call a meta-variable* X_i *a* γ*-variable.*

7. *Let* $\delta^p x\varphi(x)$ *be a signed formula of polarity* p*, and* Q' *an initial indexed formula trees with* ***Label***$(Q') = \varphi(x)$ *where* x *is a new parameter. Then*

$$Q = \begin{tikzcd} \delta^p x\varphi(x)^p_\delta \arrow[d, dash] \\ Q' \end{tikzcd}$$

is an initial indexed formula tree with ***Label***$(Q) := \delta^p x\varphi(x)$*, of polarity* p *and uniform type* δ*. The secondary type of* Q' *is* δ_0*.*
We say that Q' *is the binding position for* x*. We also call a parameter* x *a* δ*-variable.*

Example 5.1.2. *Consider the following formula*

$$(\exists\delta\forall\epsilon.|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon) \Rightarrow \exists\forall\epsilon'.|f(x)-f(a)| < \epsilon \tag{5.1}$$

whose initial indexed formula tree is shown below:

$$\begin{tikzcd}[column sep=small]
& & (\exists\delta\forall\epsilon.|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon) \Rightarrow (\forall\epsilon'.|f(x)-f(a)| < \epsilon')^+_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\
& (\exists\delta\forall\epsilon.|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon)^-_\delta \arrow[d, dash] & & (\forall\epsilon'.|f(x)-f(a)| < \epsilon')^+_\delta \arrow[d, dash] \\
& (\forall\epsilon.|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon)^-_\gamma \arrow[d, dash] & & (|f(x)-f(a)| < \epsilon')^+_\circ \\
& (|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon)^-_\beta \arrow[dl, dash] \arrow[dr, dash] & & \\
|x-a| < \delta^+_\circ & & |f(x)-f(a)| < \epsilon^-_\circ &
\end{tikzcd}$$

Semantically, the validity of indexed formula trees is defined via the notion of a path, which reflects the satisfiability of formulas with respect to its constituents.

Definition 5.1.3 (Paths)**.** *Let* Q *be an indexed formula tree. Then a* path *in* Q *is a sequence* $\ll Q_1, \ldots, Q_n \gg$ *of* α*-related nodes in* Q*. The sets* $\mathcal{P}(Q)$ *of paths through* Q *is the smallest set containing* $\{\ll Q \gg\}$ *and which is closed under the following operations:*

α**-Decomposition:** *If* Q' *is a node of primary type* α *and subtrees* Q_1, Q_2*, and* $P \cup \{\ll \Gamma, Q' \gg\} \in \mathcal{P}(Q)$*, then* $P \cup \{\ll \Gamma, Q_1, Q_2 \gg\} \in \mathcal{P}(Q)$*.*

β**-Decomposition:** *If* Q' *is a node of primary type* β *and subtrees* Q_1, Q_2*, and* $P \cup \{\ll \Gamma, Q' \gg\} \in \mathcal{P}(Q)$*, then both* $P \cup \{\ll \Gamma, Q_1 \gg\} \in \mathcal{P}(Q)$ *and* $P \cup \{\ll \Gamma, Q_2 \gg\} \in \mathcal{P}(Q)$*.*

γ**-Decomposition:** *If* Q' *is a node of primary type* γ *and subtrees* $Q_1, \ldots, Q_n$*, and* $P \cup \{\ll \Gamma, Q' \gg\} \in \mathcal{P}(Q)$*, then* $P \cup \{\ll \Gamma, Q_1, \ldots, Q_n \gg\} \in \mathcal{P}(Q)$*.*

δ**-Decomposition:** *If* Q' *is a node of primary type* δ *and subtree* Q'*, and* $P \cup \{\ll \Gamma, Q' \gg\} \in \mathcal{P}(Q)$*, then* $P \cup \{\ll \Gamma, Q' \gg\} \in \mathcal{P}(Q)$*.*

A common operation is to replace meta-variables by actual terms within a proof. The "occurs-in-check" is realized as an acyclicity check of a directed graph obtained from the structure of the indexed formula tree and additional edges between binding nodes of the instantiated meta variable and its occurring parameters. This check corresponds to the eigenvariable condition in the sequent calculus and ensures that eigenvariables have to be introduced before they are assigned to meta-variables. A similar realization can be found in [KO99].

Definition 5.1.4 (Structural Ordering)**.** *Let Q be an indexed formula tree. The* structural ordering $\prec_Q$ *is a binary relation among the nodes in Q defined by: $Q_1 \prec_Q Q_2$ iff Q_1 dominates Q_2 in Q.*

Definition 5.1.5 (Quantifier Ordering)**.** *Let Q be an indexed formula tree and σ an (idempotent) substitution for meta-variables bound on γ_0-type positions in Q by terms containing only meta-variables and parameters bound in Q. The* quantifier ordering $\prec_V$ *induced by σ is the binary relation defined by: $Q_0 \prec_V Q_1$ iff there is an $X \in dom(\sigma)$ bound on Q_1 and there occurs in $\sigma(X)$ a parameter bound on Q_0.*

Combining both relations results in the reduction relation, which needs to be acyclic for a substitution σ in order to be applicable.

Definition 5.1.6 (Reduction-Relation $\lhd$)**.** *Let Q be an indexed formula tree and σ be a variable substitution. The* reduction relation $\lhd$ *is the transitive closure of the union of $\prec_Q$ and $\prec_V$, i.e. $\lhd \ := \ (\prec_Q \cup \prec_V)^+$.*

Definition 5.1.7 (Admissible Substitutions)**.** *Let Q be an indexed formula tree, σ a substitution, and $\lhd$ the respective reduction relation. σ is* admissible*, if and only if*

$$\lhd\,(:= (\prec_Q \cup \prec_V)^+) \tag{5.2}$$

is irreflexive.

Example 5.1.8. *Consider the formula $(\exists x_\iota.\forall y_\iota.p(x) \Rightarrow p(y)$, which is invalid (e.g., consider p to be the even predicate and let x, y range over the natural numbers). Thus, it should not be possible to instantiate x with y, which would close the proof. The indexed formula tree is shown below:*

$$\begin{tikzcd}
 & (\exists x_\iota.\forall y_\iota.p(x) \Rightarrow p(y))^+_\gamma \arrow[d, dash] & \\
 & (\forall y_\iota.p(x) \Rightarrow p(y))^+_\delta \arrow[d, dash] & \\
 & (p(x) \Rightarrow p(y))^+_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\
p(x)^-_\circ & & p(y)^+_\circ
\end{tikzcd}$$

We label the nodes top down and from left to right by $n_1, \ldots, n_4$. The quantifier ordering is induced by the substitution $\sigma = \{x \mapsto y\}$ is $n_2 \to n_1$, which results, together with the structural ordering induced by the tree, in the following graph, which is cyclic. Hence the substitution is not admissible.

$$\begin{tikzcd}
 & n_1 \arrow[d, bend left] & \\
 & n_2 \arrow[u, bend left] \arrow[dl] \arrow[dr] & \\
n_3 & & n_4
\end{tikzcd}$$

5.1.1 Instantiations

Core provides a rule to apply admissible substitutions to indexed formula trees.

Definition 5.1.9 (Instantiation of Indexed Formula Trees)**.** *Let Q be an indexed formula tree and X a γ-variable that occurs free in $\textbf{\textit{Label}}(Q)$. The instantiation of X by t in Q is defined as*

- *if Q is a leaf node, then we replace Q by an initial indexed formula tree for $\{t/X\}(\textbf{\textit{Label}}(Q))$.*
- *Otherwise, apply $\{t/X\}$ to the label of Q, and recursively apply it to the subnodes of Q.*

Definition 5.1.10 (Substitution Application on Indexed Formula Trees)**.** *Let Q be an indexed formula tree, σ its actual substitution, and σ' a new substitution. If σ' is applicable on Q with σ, then we apply σ' to Q. The result of the substitution application is the (instantiated) indexed formula tree together with the new substitution $\sigma' \circ \sigma$.*

Example 5.1.11. *Take as an example the indexed formula tree for the positive formula*

$$(\forall p_{\iota\to o}.\forall q_{\iota\to o}.\exists r_{\iota\to o}.\forall x_\iota.(p(x) \lor q(x)) \Rightarrow r(x))^+.$$

The initial indexed formula tree is viewed on the left-hand side of Figure 5.1 and the actual substitution is the empty substitution. Instantiation of the γ-variable $R_{\iota\to o}$ with $\lambda y_\iota.p(y) \lor q(y)$ results in the indexed formula tree on the right-hand side below. Note how the leaf node $R(x)^\circ_+$ in the initial indexed formula tree is replaced by an initial indexed formula tree for the positive $\beta\eta$-normalized formula $(\{\lambda y_\iota.P(y) \lor Q(y)/R\}R(x)))^+$.

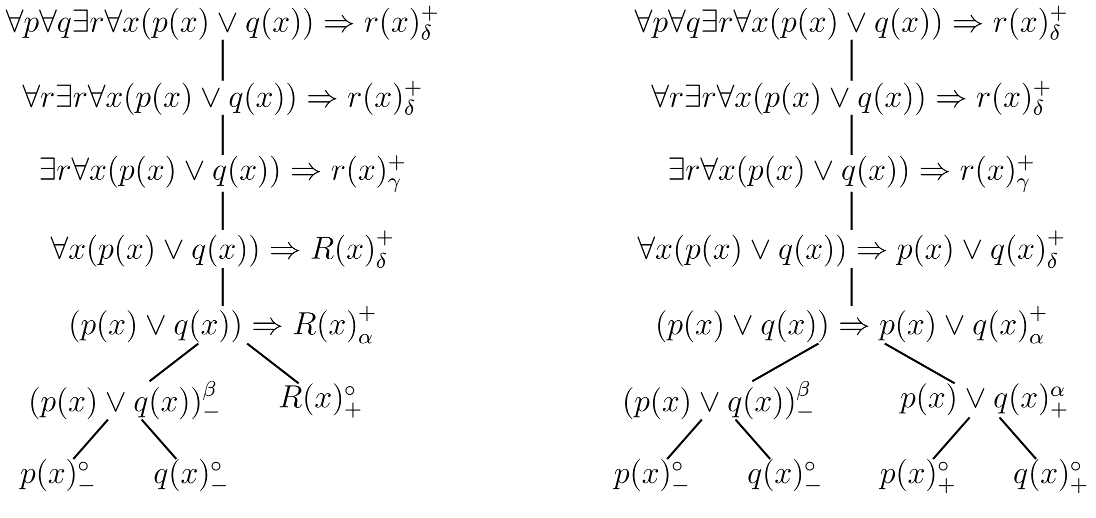

$\forall p\forall q\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\delta$
$\forall r\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\delta$
$\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\gamma$
$\forall x(p(x) \lor q(x)) \Rightarrow R(x)^+_\delta$
$(p(x) \lor q(x)) \Rightarrow R(x)^+_\alpha$
$(p(x) \lor q(x))^\beta_-$
$R(x)^\circ_+$
$p(x)^\circ_-$
$q(x)^\circ_-$
$\forall p\forall q\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\delta$
$\forall r\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\delta$
$\exists r\forall x(p(x) \lor q(x)) \Rightarrow r(x)^+_\gamma$
$\forall x(p(x) \lor q(x)) \Rightarrow p(x) \lor q(x)^+_\delta$
$(p(x) \lor q(x)) \Rightarrow p(x) \lor q(x)^+_\alpha$
$(p(x) \lor q(x))^\beta_-$
$p(x) \lor q(x)^\alpha_+$
$p(x)^\circ_-$
$q(x)^\circ_-$
$p(x)^\circ_+$
$q(x)^\circ_+$

Figure 5.1: Indexed formula tree before and after the application of the instantiation rule

5.1.2 Core Expansion Rules

Core comes along with three rules to handle equality and equivalences: Leibniz equality, functional extensionality, and an expansion rule for equivalences. These rules introduce new nodes in the indexed formula tree as described below:

Leibniz' Equality

Generally, extensionality refers to the principle to judge objects to be equal if they have the same external properties. The rule for Leibniz equality states that two terms are equal iff they share the same properties, i.e.,

$$x =^\tau y \equiv \forall P : \tau \to o. P\ x \Rightarrow P\ y \tag{5.3}$$

It can be applied to terms of the form $(s \Leftrightarrow t)^p$ and $(s = t)^p$ with $p \in \{+, -\}$ and introduces the right-hand side of (5.3).

Definition 5.1.12 (Leibniz' Equality Introduction)**.** *Let Q_e be a leaf node in some indexed formula of polarity p, uniform type $e \in \{\epsilon, \zeta\}$, such that* ***Label***$(Q_e) = \epsilon(s_\tau, t_\tau)$ *or* ***Label***$(Q_e) = \zeta(s_\tau, t_\tau)$*. Further let Q_L be an initial indexed formula tree for the signed formula $(\forall P_{\tau \to o} P(s) \Rightarrow P(t))^p$. Then we can replace Q_e by*

$$Q_{Leibniz} = \begin{tikzcd} & \boldsymbol{Label}(Q_e)^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_e & & Q_L \end{tikzcd}$$

We call the new node a Leibniz *node.*

Functional Extensionality

Functional extensionality states that two functions that are equal have equal results for all possible inputs. The rule requires a specific locality to ensure its soundness, namely γ-locality (resp. δ-locality). The underlying condition is that it is only possible to abstract over meta-variables bound in some γ-type node (δ-type node) and that it must be possible to move its quantifier in front of $\epsilon(s(X), t(X))$ $(\zeta(s(x), t(x)))$.

Definition 5.1.13 (Local Variables)**.** *Given an indexed formula tree Q and a node Q' inside Q whose label contains a free variable x. If x is bound in some γ-type node in Q, then x is* γ-local for *Q' iff Q' is the binding position for x or Q' has a direct parent node Q'' such that*

- *Q'' is of primary type β, x does not occur in the label of the sibling of Q', and x is γ-local for Q'', or*

- *Q' is of either primary type α or γ and x is γ-local for Q''.*

The dual property of a δ-local variable x for some Q' is defined for variables bound in a δ-type node and holds iff Q' is the binding node for x or Q' has a direct parent node Q'' such that

- *Q'' is of type α, x does not occur in the label of the sibling of Q', and x is δ-local for Q'', or*

- *Q'' is of either type β or δ and x is δ-local for Q''.*

Making use of this definition, the extensionality introduction rule can now be stated as follows:

Definition 5.1.14 (Extensionality Introduction)**.** *Let Q_e be a leaf in some indexed formula of polarity p, uniform type $e \in \{\epsilon, \zeta\}$, such that $\boldsymbol{Label}(Q_e) = \overset{\epsilon}{\zeta}(s,t)$. Let further be x a variable that is* local for *Q_e, and Q_{Ext} be an initial indexed formula tree for the signed formula $(\lambda x.s = \lambda x.t)^p$. Then we can replace Q_e by*

$$Q_{Ext-I} = \begin{tikzcd} & \boldsymbol{Label}(Q_e)^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_e & & Q_{Ext} \end{tikzcd}$$

We call the new node an Extensionality introduction *node.*

Boolean Extensionality

The rule of Boolean extensionality replaces positive equations and equivalences over formulas A_o and B_0 into $((A \Rightarrow B) \wedge (B \Rightarrow A))^+$. As a result, polarities and uniform types can be assigned to subformulas, having the effect that subsequent resolution replacement rules can also modify the subformulas.

Definition 5.1.15 (Boolean ζ-Expansion)**.** *Let Q_ζ be a leaf node in some indexed formula tree of positive polarity, uniform type ζ, and label $\zeta(A,B)$, where A and B are of type o. Let further be Q_E be an initial indexed formula tree for the signed formula $((A \Rightarrow B) \wedge (B \Rightarrow A))^+$. Then we can replace Q_ζ by*

$$Q_{\zeta Expansion} = \begin{tikzcd} & \boldsymbol{Label}(Q_\zeta)^+_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_\zeta & & Q_E \end{tikzcd}$$

We call the new node a ζ-expansion node.

5.1.3 Increasing Multiplicities

It may become necessary to create multiple instances of a subformula to be able to construct a proof. For example, in the sequent calculus, two copies of the subformula $\forall x.Q(x)$ are needed to construct a proof of $\forall x.Q(x) \Rightarrow Q(a) \wedge Q(b)$. The number of copies is known as *multiplicity*, and the problem of finding this number is what makes first-order logic undecidable (although semidecidable). Many automatic provers which are based on tableau or the connection method work by performing an iterative depth-first search on the maximal number of allowed multiplicities. CORE provides a special rule to dynamically increase the multiplicity of a given subformula, which is beneficial in the interactive setting where the multiplicity cannot be fixed in advance. Essentially, the rule copies parts of the indexed formula tree, thereby taking modifications by extensionality rules into account. To understand the rule, observe the following:

- Increasing the multiplicity of one quantifier may require to increase the multiplicity of others
- It is sufficient to increase only the multiplicities of the quantifiers that are smallest with respect to the structural ordering, as the multiplicity of the quantifiers below are increased implicitly.

The following two definitions capture the minimal set of nodes that need to be copied when increasing the multiplicity.

Definition 5.1.16 (Convex Set of Subtrees). *Let Q be an indexed formula tree with admissible substitution σ, $\mathcal{K}$ a set of independent*[1] *subtrees of Q.*

Then $\mathcal{K}$ is convex with respect to σ *iff for all $Q \in \mathcal{K}$ and for all γ-, δ-, -variables x bound in Q we have: if x occurs in some instance $\sigma(y)$ for some y, then there exists some $Q' \in \mathcal{K}$ in which y is bound.*

Definition 5.1.17 (Determining Nodes to Increase Multiplicities). *Let Q be an indexed formula tree, and σ an admissible substitution. Let Q_m be a node of secondary type γ_0. The subtrees to copy in order to increase the multiplicity of Q_m's parent are given by the predicate $\mu(Q_m)$ that is inductively defined:*

$$\mu(Q_m) = \{Q_m\} \quad \cup \left(\bigcup_{Q' | Q_m \prec Q'} \mu(Q')\right) \cup \left(\bigcup_{Q' \in Inst_Q(Q_m)} \mu(Q')\right)$$

where $Inst_Q(Q_m)$ is the set of binding nodes of variables x, such that y occurs in $\sigma(x)$ and y is bound on Q_m. If Q_m is not a binding node, then $Inst_Q(Q_m) = \emptyset$. We denote by $\mu(Q_m)_{min}$ the subset of the minimal nodes with respect to $\lhd$ of $\mu(Q_m)$.

Being able to determine the minimal set of nodes that need to be copied allows the definition of the following rule to dynamically increase the multiplicity:

Definition 5.1.18 (Multiplicity Increase). *Let Q be an indexed formula tree with actual admissible substitution σ. Furthermore let Q_m be a node of secondary type γ_0. In order to increase the multiplicity of Q_m's parent we determine the set $\mu(Q_m)_{min}$. For each $Q' \in \mu(Q_m)_{min}$*

1. *we copy Q' to obtain Q'' together with a variable renaming ρ and an isomorphic mapping from Q' to Q''. Subsequently we add Q'' to the parent node of Q';*

2. *we extend the variable substitutions of σ by*

$$\{\rho(\sigma_Q(x))/\rho(x) \mid x \in dom(\rho)\}$$

5.2 Free Variable Indexed Formula Trees

A free variable indexed formula tree is defined on top of the underlying indexed formula tree. It is a free variable representation of the indexed formula tree which is transformed by the calculus rules and used to represent the proof state to the user. Initially, the free variable indexed formula tree is a representation of the initial indexed formula tree without quantifier:

Definition 5.2.1 (Initial Free Variable Indexed Formula Trees). *We define* initial free variable indexed formula trees *R inductively over the structure of some given indexed formula tree Q. Each node of the tree has a formula as label, a polarity, a uniform type, and possibly the indexed formula tree node for which it is a working copy.*

1. *If $Q = A^p_\circ$ is a literal node, then $R = {}^p_\circ A_Q$ is a free variable indexed formula tree of the same label, polarity and uniform type than Q and a reference to Q. They are leaves of free variable indexed formula trees.*

[1] i.e. no nested subtrees.

2. *If $Q = \epsilon(s,t)^p_\epsilon$, then $R = {}^p_\epsilon\epsilon(s,t)_Q$ is a free variable indexed formula tree. They are leaves of free variable indexed formula trees.*

3. *If*

$$Q = \begin{tikzcd} \alpha(\textbf{\textit{Label}}(Q'))^p_\alpha \arrow[d, dash] \\ Q' \end{tikzcd}$$

is an indexed formula tree and R' is a free variable indexed formula tree for Q', then

$$R = \begin{tikzcd} {}^p_\alpha\alpha(\textbf{\textit{Label}}(R')) \arrow[d, dash] \\ R' \end{tikzcd}$$

is a free variable indexed formula tree for Q.

4. *If*

$$Q = \begin{tikzcd} & \alpha(\textbf{\textit{Label}}(Q_1), \textbf{\textit{Label}}(Q_2))^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_1 & & Q_2 \end{tikzcd}$$

is an indexed formula tree, R_1 and R_2 are free variable indexed formula trees for Q_1 and Q_2 respectively, then

$$R = \begin{tikzcd} & {}^p_\alpha\alpha(\textbf{\textit{Label}}(R_1), \textbf{\textit{Label}}(R_2)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_1 & & R_2 \end{tikzcd}$$

is a free variable indexed formula tree for Q.

5. *If*

$$Q = \begin{tikzcd} & \beta(\textbf{\textit{Label}}(Q_1), \textbf{\textit{Label}}(Q_2))^p_\beta \arrow[dl, dash] \arrow[dr, dash] & \\ Q_1 & & Q_2 \end{tikzcd}$$

is an indexed formula tree, R_1 and R_2 are free variable indexed formula trees for Q_1 and Q_2 respectively, then

$$R = \begin{tikzcd} & {}^p_\beta\beta(\textbf{\textit{Label}}(R_1), \textbf{\textit{Label}}(R_2)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_1 & & R_2 \end{tikzcd}$$

is a free variable indexed formula tree for Q.

6. *If*

$$Q = \begin{tikzcd} & \gamma^p x\varphi(x)^p_\gamma \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ Q_1 & \ldots & Q_n \end{tikzcd}$$

is an indexed formula tree and $R_1, \ldots, R_n$ are free variable indexed formula trees for $Q_1, \ldots, Q_n$ respectively, then let $R'_1 := R_1$, and

$$R'_{i+1} := \begin{tikzcd} & {}^p_\alpha\alpha(\textbf{\textit{Label}}(R'_i), \textbf{\textit{Label}}(R_{i+1})) \arrow[dl, dash] \arrow[dr, dash] & \\ R'_i & & R_{i+1} \end{tikzcd}$$

for $1 \leq i \leq (n-1)$. Then $R := R'_n$ is a free variable indexed formula tree for Q. Note that the R'_i do not have a reference to Q.

7. *If*

$$Q = \begin{tikzcd} \delta^p x \varphi(x)^p_\delta \arrow[d, dash] \\ Q' \end{tikzcd}$$

is an indexed formula tree and R is a free variable indexed formula tree for Q', then R is also a free variable indexed formula tree for Q.

Example 5.2.2. *For the initial indexed formula tree of example 5.1.2, the initial free variable indexed formula tree looks as follows:*

$$\begin{tikzcd}[column sep=small] & (|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon) \Rightarrow \epsilon'.|f(x)-f(a)| < \epsilon')^+_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ (|x-a| < \delta \Rightarrow |f(x)-f(a)| < \epsilon)^-_\beta \arrow[d, dash] \arrow[dr, dash] & & (|f(x)-f(a)| < \epsilon')^+_\circ \\ |x-a| < \delta^+_\circ & |f(x)-f(a)| < \epsilon^-_\circ & \end{tikzcd}$$

Definition 5.2.3 (Equality Free Variable Indexed Formula Trees)**.** *Let R, R' be two free variable indexed formula trees. We say that R and R' are α-equal iff their labels are equal up to the renaming of bound variables.*

A free variable indexed formula tree is proved if it has been transformed to trivially valid formula. Formally, this is captured by the following definition:

Definition 5.2.4 (Proved and Disproved Free Variable Indexed Formula Trees)**.** *Let R be a literal free variable indexed formula tree. Then*

- *R is* proved*, iff it has either negative polarity and its label is* false*, or it has positive polarity and its label is* true*, or it is of primary type a ζ and the label is $\zeta(t,t)$.*
- *R is* disproved*, iff either it has positive polarity and its label is* false*, or it has negative polarity and its label is* true*, or it is of primary type ϵ and the label is $\epsilon(t,t)$.*

Let R be a free variable indexed formula tree. R is proved (resp. disproved), iff

- *R is a proved (resp. disproved) literal free variable indexed formula tree,*
- *or R is of primary type α (resp. β) and some subtree is proved (resp. disproved),*
- *or R is of primary type β (resp. α) and all subtrees are proved (resp. disproved),*

Again, the underlying notation is the notion of paths, which carries over from indexed formula trees:

Definition 5.2.5 (Paths in Free Variable Indexed Formula Trees)**.** *Let R be a free variable indexed formula tree. A path in R is a sequence $\ll R_1, \ldots, R_n \gg$ of α-related nodes in R. The sets $\mathcal{P}(R)$ of paths through R is the smallest set containing $\{\ll R \gg\}$ and which is closed under the following operations:*

α-Decomposition: *If R' is a node of primary type α and subtrees R_1, R_2, and $P \cup \{\ll \Gamma, R' \gg\} \in \mathcal{P}(R)$, then $P \cup \{\ll \Gamma, R_1, R_2 \gg\} \in \mathcal{P}(R)$.*

β-Decomposition: *If R' is a node of primary type β and subtrees R_1, R_2, and $P \cup \{\ll \Gamma, R' \gg\} \in \mathcal{P}(R)$, then both $P \cup \{\ll \Gamma, R_1 \gg\} \in \mathcal{P}(R)$ and $P \cup \{\ll \Gamma, R_2 \gg\} \in \mathcal{P}(R)$.*

We are now in the position of defining a Core proof state, which consists of an indexed formula tree, its free variable working copy and an actual substitution:

Definition 5.2.6 (Proof State, Soundness & Safeness). *Let Q an indexed formula tree, σ a substitution, and let R be a free variable indexed formula tree. Then a* proof state *is denoted by $[Q, \sigma \triangleright R]$. A proof step is a transformation of some proof state $[Q, \sigma \triangleright R]$ into another proof state $[Q', \sigma' \triangleright R']$, which is denoted as $[Q, \sigma \triangleright R] \longmapsto [Q', \sigma' \triangleright R']$.*

Such a proof step is sound *iff σ is admissible with respect to Q and there is a satisfiable path in R then σ' is admissible with respect to Q' and there is a satisfiable path in R'.*

A proof step is safe *iff σ' is admissible with respect to Q' and there is an satisfiable path in R' then σ is admissible with respect to Q and there is an satisfiable path in R.*

Subsequently, we introduce the Core calculus rules which modify the proof state, which consist of the following twelve rules: contraction and weakening, simplification, Leibniz equality, functional and Boolean extensionality, instantiation, increase of multiplicities, rewriting and resolution replacement rules, and the cut rule. All rules are proven to be sound in [Aut03]. Within our setting, the resolution replacement rules are of particular interest.

5.2.1 Replacement Rules

As we have seen in Section 4.1.4, uniform notation allows us to determine the logical context of a subformula by determining the set of all formulas which are α-related to the subformula. From the context, so-called *replacement rules* of the form $u \to \langle v_1, \ldots, v_n \rangle$ can be derived, which allow the replacement of a subtree denoted by u by a newly generated subtree $\overline{\beta}(v_1, \ldots, v_n)$. Core provides two kinds of replacement rules: *resolution replacement rules* and *rewriting replacement rules.* Resolution replacement rules are rules where the left-hand side is a subformula with a polarity. Rewrite replacement rules stem from negative equalities and negative equivalences.

Resolution Replacement Rules

In order to formalize the notion of a replacement rule, we first define the *conditions* of some subtree as the formal characterization of the β-related formulas of some node.

Definition 5.2.7 (Weakening of Free Variable Indexed Formula Trees). *Let R be a free variable indexed formula tree. The set $\mathcal{W}(R)$ of weakened free variable indexed formula trees for R is defined recursively over the structure of R:*

$$\begin{aligned}
\mathcal{W}(R) &= \{R\} \text{ iff } R \text{ is a literal node} && (5.4)\\
\mathcal{W}(\alpha^p(R_1, R_2)) &= \{\alpha^p(R_1^w, R_2^w) \mid R_i^w \in \mathcal{W}(R_i), i = 1, 2\} && (5.5)\\
&\quad \cup \mathcal{W}(R_1) \cup \mathcal{W}(R_2) && (5.6)\\
\mathcal{W}(\beta^p(R_1, R_2)) &= \{\beta^p(R_1^w, R_2^w) \mid R_i^w \in \mathcal{W}(R_i), i = 1, 2\} && (5.7)
\end{aligned}$$

Definition 5.2.8 (Node Conditions). *Let R, c be nodes in some free variable indexed formula tree, such that c is a parent node of R. Let $R_1, \ldots, R_n$ be all maximal nodes that are below c and β-related to R. Then the conditions of R are given by the set $\mathcal{C}_R^c := \mathcal{W}(R_1) \times \ldots \times \mathcal{W}(R_n)$.*

Definition 5.2.9 (Admissible Resolution Replacement Rules)**.** *Let R_0, R be nodes in some free variable indexed formula tree and σ the actual overall substitution. Then $R_0 \to \langle R_1, \dots, R_n\rangle$ is an* admissible resolution replacement rule *for R, iff*

1. *R_0 and R have opposite polarities and are α-related by a node c,*
2. *and $(R_1, \dots, R_n) \in \mathcal{C}^c_{R_0}$.*

Definition 5.2.10 (Resolution Replacement Rule Application)**.** *Let $[Q, \sigma \rhd R]$ be a proof state, a a node in R, and $u \to \langle v'_1, \dots, v'_n\rangle$ ($n \geq 0$) an admissible resolution replacement rule for a (i.e. $v'_i \in \mathcal{W}(v_i)$, v_i β-related to u) such that u and a are* connectable, *i.e. if they are equal and have opposite polarity. The application of $u \to \langle v'_1, \dots, v'_n\rangle$ to a is defined as follows:*

- *For each v'_i, we determine the node p_i which governs a and and β-insert v'_i on p_i.*
- *subsequently, we replace the subtree a by an initial free variable indexed formula tree for* true$^+$*, if a has positive polarity, or otherwise for* false$^-$*.*

Example 5.2.11. *Consider the formula*

$$\left(\left[(A_1^- \wedge^\alpha B_1^-)^- \wedge^\alpha ([A_2^+ \wedge^\beta B_2^+]^+ \Rightarrow^\beta C_1^-)^-\right]^- \Rightarrow^\alpha C_2^+\right)^+ \tag{5.8}$$

from which we can generate the replacement rule $C_1^- \to \langle A_2^+, B_2^+\rangle$ for C_2^+. We can apply this replacement rule to rewrite 5.8 to

$$\left(\left[(A_1^- \wedge^\alpha B_1^-)^- \wedge^\alpha ([A_2^+ \wedge^\beta B_2^+]^+ \Rightarrow^\beta C_1^-)^-\right]^- \Rightarrow^\alpha A_2^+ \wedge B_2^+\right)^+ \tag{5.9}$$

From this formula we can generate the replacement rule $\langle (A_1 \wedge B_1)^- \to true^+\rangle$ and transform 5.9 to

$$\left(\left[(A_1^- \wedge^\alpha B_1^-)^- \wedge^\alpha ([A_2^+ \wedge^\beta B_2^+]^+ \Rightarrow^\beta C_1^-)^-\right]^- \Rightarrow^\alpha true^+\right)^+ \tag{5.10}$$

Rewriting Replacement Rules

Definition 5.2.12 (Admissible Rewriting Replacement Rules)**.** *Let R_0, R be nodes in some free variable indexed formula tree, R_0 of primary type ϵ and label $\epsilon(s,t)$, and σ the actual overall substitution. Then $s \to \langle t, R_1, \dots, R_n\rangle$ and $t \to \langle s, R_1, \dots, R_n\rangle$ are* admissible rewriting replacement rules *for R, iff*

1. *R_0 and R are α-related by a node c,*
2. *and it holds that $(R_1, \dots, R_n) \in \mathcal{C}^c_{R_0}$.*

Definition 5.2.13 (Rewriting Replacement Rule Application On Nodes)**.** *Let $[Q, \sigma \rhd R]$ be a proof state, a a node in R of polarity p and let $u \to \langle v, v'_1, \dots, v'_n\rangle$ ($n \geq 0$) be an admissible rewriting replacement rule for a, where u and v are the left- and right-hand sides of an ϵ-type position v_0. The application of $u \to \langle v, v'_1, \dots, v'_n\rangle$ to a is defined as follows:*

1. *Apply the Leibniz' equality introduction rule to v_0 to obtain*

$$\beta^-(P(\boldsymbol{Label}(v))^p, P(\boldsymbol{Label}(u))^{-p}),$$

2. *Instantiate P by $\lambda x.x$ to obtain $\beta^{-}(\boldsymbol{Label}(v)^{p}, \boldsymbol{Label}(u)^{-p})$, which results in the resolution replacement rule*

$$\boldsymbol{Label}(u)^{-p} \to \langle \boldsymbol{Label}(v)^{p}, v_1, \ldots, v_n\rangle.$$

3. *Apply $\boldsymbol{Label}(u)^{-p} \to \langle \boldsymbol{Label}(v)^{p}, v_1, \ldots, v_n\rangle$ to a.*

Definition 5.2.14 (Rewriting Replacement Rule Application Inside Literal Nodes)**.** *Let $[Q, \sigma \triangleright R]$ be a proof state, a a literal node of label φ in R and of polarity p, π a valid subterm occurrence inside $\boldsymbol{Label}(a)$, and let $u \to \langle v, v'_1, \ldots, v'_n\rangle$ $(n \geq 0)$ be an admissible rewriting replacement rule for a, where u and v are the left- and right-hand sides of an ϵ-type position v_0 of label $\epsilon^{-}(s,t)$. The application of $u \to \langle v, v'_1, \ldots, v'_n\rangle$ on a at π is defined as follows:*

- *Let $x_1, \ldots, x_n$ be the variables that are free in $\varphi_{|\pi}$, but not in φ. Let further be σ' a substitution such that $\sigma'(s) = \sigma'(\varphi_{|\pi})$ and $x_i \notin dom(\sigma'), 1 \leq i \leq n$.*
- *Let $D := \{X \in dom(\sigma') \mid \exists x_i . x_i \in \sigma'(X)\}$ be the variables that are instantiated with a term in which occurs one of the x_i. Then σ' is partitioned into two disjunct substitutions defined by*

$$\sigma'_1 := \sigma'_{|D} \text{ and } \sigma'_2 := \sigma'_{|dom(\sigma')\setminus D}$$

- *Apply the extensionality introduction rule on v_0 for the variables in D to obtain v'_0 of label $\lambda y_1 . \ldots \lambda y_n s = \lambda y_1 . \ldots \lambda y_n . t$.*

  If this fails the rule application fails.
- *Otherwise apply the Leibniz' equality introduction on v'_0 to obtain the formula*

$$\gamma^{p} P \beta^{p}(P(\lambda y_1 . \ldots \lambda y_n . s)^{-p}, P(\lambda y_1 . \ldots \lambda y_n . t)^{p}).$$

  This results in the resolution replacement rule

$$P(\lambda y_1 \ldots \lambda y_n s)^{-p} \to \langle P(\lambda y_1 \ldots \lambda y_n t)^{p}, v_1, \ldots, v_n\rangle.$$

- *Apply the substitution $\{\lambda f . \varphi_{|\pi \leftarrow f(\sigma'_1(y_n), \ldots, \sigma'_1(y_1))}/P\} \circ \sigma'_2$.*
- *Apply the (instantiated) resolution replacement rule.*

Because the definition is rather technical and involves several steps we illustrate the rewriting below binders by means of an example.

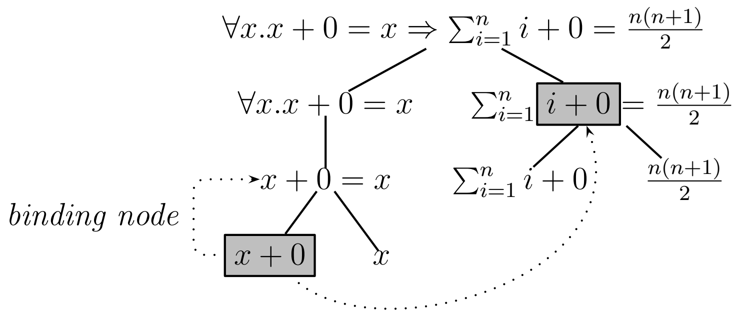
Example 5.2.15. *Suppose the following definition of the operator* sum*:*

$$\text{sum} : (\text{nat} \to \text{nat}) \to \text{nat} \to \text{nat} \tag{5.11}$$
$$\text{sum} f\ 0 = 0 \tag{5.12}$$
$$\text{sum} f\ suc(k) = f\ suc(k) \ + \ \text{sum}\ f\ k \tag{5.13}$$

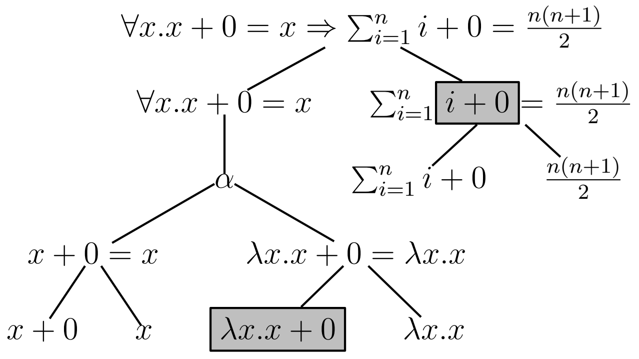
For example, having $f = \lambda x.x$, the definition works as follows:

$$\text{sum}\, f\, \text{suc}(\text{suc}(0)) = f\ 2 \ + \ \text{sum}\ f\ \text{suc}(0) \tag{5.14}$$
$$= f\ 2 \ + \ f\ 1 \ + \ \text{sum}\ f\ 0 \tag{5.15}$$
$$= f\ 2 \ + \ f\ 1 \ + 0 \tag{5.16}$$

We will write sum f *in the more intuitive form as $\sum_{i=1}^{n} f$. Moreover, we suppose to have the rewrite rule $x + 0 = x$, which can then be used to rewrite $\sum_{i=1}^{n} i + 0 = \frac{n(n+1)}{2}$ to $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. For our example, the indexed formula tree is shown below.*


$\forall x.x+0=x \Rightarrow \sum_{i=1}^n i+0=\frac{n(n+1)}{2}$
$\forall x.x+0=x$
$\sum_{i=1}^n \boxed{i+0} = \frac{n(n+1)}{2}$
$x+0=x$
$\sum_{i=1}^n i+0$
$\frac{n(n+1)}{2}$
binding node
$\boxed{x+0}$
x

We consider the substitution $\{x \mapsto i\}$. The variables which are free in $i+0$ but not free in the overall node are i, thus $x_1 = i$ and $D = \{x\}$. Moreover, x is γ-local. Therefore, extensionality introduction is applicable and we introduce a new node that contains $\lambda x.x+0 = \lambda x.x$. This results in the following indexed formula tree:


$\forall x.x+0=x \Rightarrow \sum_{i=1}^n i+0=\frac{n(n+1)}{2}$
$\forall x.x+0=x$
$\sum_{i=1}^n \boxed{i+0} = \frac{n(n+1)}{2}$
$\sum_{i=1}^n i+0$
$\frac{n(n+1)}{2}$
α
$x+0=x$
$\lambda x.x+0=\lambda x.x$
$x+0$
x
$\boxed{\lambda x.x+0}$
$\lambda x.x$

However, in general the locality condition is a very strong condition. As a consequence, conditional rewrites such as $x \neq 0 \Rightarrow x \cdot x^{-1} = 1$ cannot directly be applied below binders as rewrite replacement rules, as in this case x is not γ-local. Generally, these implicative rewrites are of the form

$$A_1 \wedge \ldots \wedge A_n \Rightarrow s = t \tag{5.17}$$

or equivalently

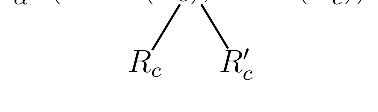
$$A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow s = t \tag{5.18}$$

where A_i are of type o. Possible solutions are the use of a choice operator or the use of an if-then-else construct.

5.2.2 Contraction, Weakening and Cut

Similar to the sequent calculus, CORE provides a contraction and a weakening rule to copy a formula or to remove assumptions from the proof state. Beside the standard use of the cut rule, this rule is used to introduce new variables to the proof state, which can arise as result of higher-order unification.

Definition 5.2.16 (Contraction Rule)**.** *Let $[Q, \sigma \rhd R]$ be a proof state, R_c a subtree of polarity p in R, and R_c' a copy of R_c. The application of the contraction rule results in the proof state $[Q, \sigma \rhd R']$, where R' is obtained from R by replacing the subtree R_c by*


${}^p_\alpha\alpha(\textbf{\textit{Label}}(R_c), \textbf{\textit{Label}}(R_c'))$
R_c
R_c'

Definition 5.2.17 (Weakening Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, R_w a subtree in R, and $R'_w \in \mathcal{W}(R_w)$. The application of the* weakening rule *results in a proof state $[Q, \sigma \triangleright R']$, where R' is obtained from R by replacing the subtree R_w by R'_w.*

Definition 5.2.18 (Cut Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, and let R' be a subtree of R with polarity p and label φ, and A a formula. Assume further, that Q' is the smallest subtree of Q that contains all subtrees referenced in R'. The cut over A on R' results in a new proof state $[Q^*, \sigma^* \triangleright R^*]$, where Q^* and σ^* result from the cut over A on Q' in Q. From there two free variable indexed formula trees R_{A^p} and $R_{A^{-p}}$ of respective signed labels A^p and A^{-p} are constructed from the initial indexed formula trees for A^p and A^{-p}. Finally R^* is obtained from R by replacing the subtree R' with the subtree*

$$\begin{tikzcd}
& & & {}^p_\beta\beta^p(\alpha^p(A^p,\varphi^p),\alpha^p(A^{-p},\varphi^p)) \arrow[dll, dash] \arrow[drr, dash] & & & \\
& {}^p_\alpha\alpha^p(A^p,\varphi^p) \arrow[dl, dash] \arrow[dr, dash] & & & & {}^p_\alpha\alpha^p(A^{-p},\varphi^p) \arrow[dl, dash] \arrow[dr, dash] & \\
R_{A^p} & & R' & & R_{A^{-p}} & & R''
\end{tikzcd}$$

where R'' is a copy of R'.

5.2.3 Simplification

The simplification rule simplifies a formula by removing solved branches from the free variable indexed formula tree.

Definition 5.2.19 (Simplification Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, and let R' be a subtree of R. The simplification rule consists of*

- *if R' is proved, replace R by an initial free variable indexed formula tree for* true^+ *if the polarity of R is positive, and otherwise by an initial free variable indexed formula tree for* false^-*,*

- *if R' is disproved, replace R by an initial free variable indexed formula tree for* false^- *if the polarity of R is positive, and otherwise by an initial free variable indexed formula tree for* true^+*,*

- *if R' a β-type node $\beta(R_1^{p_1}, R_2^{p_2})^p$ and not proved, but either R_1 or R_2 is proved, then*

  - *replace R' by R_i, if R_i is the non-proven subtree and $p = p_i$, or*
  - *replace R' by $\alpha(R_i^{p_i})^p$, if R_i is the non-proven subtree and $p \neq p_i$*

- *if R' an α-type node $\alpha(R_1^{p_1}, R_2^{p_2})^p$ and not disproved, but either R_1 or R_2 is disproved, then*

  - *replace R' by R_i, if R_i is the non-disproven subtree and $p = p_i$, or*
  - *replace R' by $\alpha(R_i^{p_i})^p$, if R_i is the non-disproven subtree and $p \neq p_i$*

- *Otherwise to leave R unchanged.*

5.2.4 Extensionality Rules

The extensionality rules are just variants for the free variable indexed formula tree.

Definition 5.2.20 (Leibniz' Equality Introduction Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, R_e an ϵ- or ζ-type subtree in R, Q_e its associated subtree in Q of polarity p and label $\overset{\epsilon}{\zeta}(s,t)$, and Q'_e an initial indexed formula tree for $(\forall P \mathbin{.} P(s) \Rightarrow P(t))^p$. The application of the* Leibniz' Equality Introduction rule *on R_e results in a proof state $[Q', \sigma \triangleright R']$. Thereby Q' is the result of applying the Leibniz' equality introduction rule on Q_e which consisted in replacing Q_e by*

$$Q_{Leibniz} = \begin{tikzcd}[column sep=tiny] & \boldsymbol{Label}(Q_e)^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_e & & Q'_e \end{tikzcd}$$

Furthermore R' is the result of replacing all literal nodes R_L in R that are annotated by Q_e with

$$\begin{tikzcd}[column sep=tiny] & {}^p_\alpha\alpha(\boldsymbol{Label}(R_L), \boldsymbol{Label}(R'_L)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_L & & R'_L \end{tikzcd}$$

where R'_L is an initial free variable indexed formula tree for Q'_e.

Moreover, it supports the standard (extensional) definition of functional equality, which says that two functions are (extensionally) equal, if, given the same input, they always produce the same result[2].

$$f = g \equiv \forall x. f(x) = g(x) \tag{5.19}$$

Definition 5.2.21 (Extensionality Introduction Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, R_e an ϵ- or ζ-type subtree in R, Q_e its associated subtree in Q of polarity p and label $\overset{\epsilon}{\zeta}(s,t)$ with local variable x, and Q'_e an initial indexed formula tree for $\overset{\epsilon}{\zeta}(\lambda x \mathbin{.} s, \lambda x \mathbin{.} t)$. The application of the* extensionality introduction rule *on R_e results in a proof state $[Q', \sigma \triangleright R']$. Thereby Q' is the result of applying the extensionality introduction rule on Q_e which consisted in replacing Q_e by*

$$Q_{Ext} = \begin{tikzcd}[column sep=tiny] & \boldsymbol{Label}(Q_e)^p_\alpha \arrow[dl, dash] \arrow[dr, dash] & \\ Q_e & & Q'_e \end{tikzcd}$$

Furthermore R' is the result of replacing all literal nodes R_L in R that are annotated by Q_e with

$$\begin{tikzcd}[column sep=tiny] & {}^p_\alpha\alpha(\boldsymbol{Label}(R_L), \boldsymbol{Label}(R'_L)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_L & & R'_L \end{tikzcd}$$

where R'_L is an initial free variable indexed formula tree for Q'_e.

Finally, there is a boolean expansion rule that replaces $A \Leftrightarrow B$ by $A \Rightarrow B \wedge B \Rightarrow A$:

[2]As a consequence, for example, the two functions $f(x) = 2x + 2$ and $g(x) = 2(x + 1)$ are equal, even though they are defined differently.

Definition 5.2.22 (Boolean ζ-Expansion Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, R_ζ a ζ-type subtree in R, and Q_ζ its associated subtree in Q of label $\zeta(A_o, B_o)$. The application of the* boolean ζ-expansion rule *on R_ζ results in a proof state $[Q', \sigma \triangleright R']$, where Q' is the result of applying the boolean ζ-expansion rule on Q_ζ which introduces an initial indexed formula tree Q_E for $((A \Rightarrow B) \wedge (B \Rightarrow A))^+$. Furthermore, R' is obtained by replacing all literal nodes R_L of R annotated by Q_ζ by*

$$\begin{tikzcd}[column sep=small] & {}^{p}_{\alpha}\alpha(\boldsymbol{Label}(R_L), \boldsymbol{Label}(R'_L)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_L & & R'_L \end{tikzcd}$$

where R'_L is an initial free variable indexed formula tree for Q_E.

5.2.5 Instantiation

Definition 5.2.23 (Instantiation Rule). *Let $[Q, \sigma \triangleright R]$ be a proof state, and σ' an substitution such that $\sigma' \circ \sigma$ is admissible. The instantiation rule results in a proof state $[Q', \sigma' \circ \sigma \triangleright R']$ where Q' results from Q as defined in Definition* (5.1.9) *and R' results from R by applying the substitution to the non-literal nodes and by replacing all literal nodes in R with associated literal node Q_L in Q that have been replaced in Q' by some $Q_L^{\sigma'}$ with an initial free variable indexed formula tree for $Q_L^{\sigma'}$.*

5.2.6 Increase of Multiplicities

Definition 5.2.24 (Increase of Multiplicities). *Let $[Q, \sigma \triangleright R]$ be a proof state, and $\mathcal{Q}$ a set of subtrees from Q of which to increase the multiplicities. The new proof state $[Q', \sigma' \triangleright R']$ is obtained by*

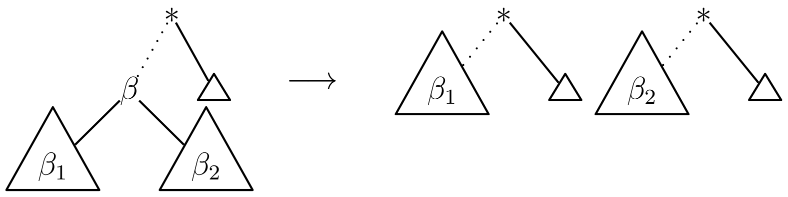
- *increasing the multiplicities in Q according to Definition* (5.1.18) *which results in Q', a variable renaming θ and a mapping ι on subtrees of the indexed formula tree Q'.*
- *Let R_M be the maximal subtrees that have an associated node of type ν_0 in $dom(\iota)$ and R_L the maximal subtrees that contain only literal nodes in $dom(\iota)$ and that do not occur in R_M. For each subtree $R_0 \in R_M \cup R_L$ we α-insert a copy of R_0 that has been renamed with respect to θ and ι.*

5.2.7 Schütte's Rule

In addition to the twelve basic calculus rules, Autexier shows in [Aut03] that the Schütte decomposition rule (see [Sch77] for details) is admissible within the CORE calculus in a generalized form. Written as an inference this rule has the following form:

$$\frac{\varphi(A^{p_A})^p \qquad \varphi(B^{p_B})^p}{\varphi(\beta(A^{p_A}, B^{p_B}))^p}\ \beta\text{-DECOMPOSE} \tag{5.20}$$

In the above equation, φ is any higher-order formula of type $o \to o$ and of the form $\lambda x_o \psi$, such that x occurs exactly once with a defined polarity in ψ^p, for any $p \in \{+, -\}$. In [Sch77] Schütte's definition of this rule is restricted to situations where φ contains no β-type formulas. However, Autexier shows in [Aut03] that the rule is sound for the general case and defines a corresponding rule for Core. The general idea of the rule is as follows:



Note that the Schütte rule internally makes use of the cut rule and also transforms the indexed formula tree.

5.3 Two Example Proofs

To illustrate the style of reasoning supported by the Core calculus, we give two example proofs. The first proof illustrates reasoning with assertions, whereas the second proof relies almost entirely on equational reasoning.

5.3.1 Simple Set Theory

Consider the simple theorem

$$\forall A, B. A \cup B = B \cup A \tag{5.21}$$

in the Core calculus, assuming the definitions

$$\forall A, B. A = B \Leftrightarrow A \subset B \wedge B \subset A \tag{5.22}$$

$$\forall A, B. A \subset B \Leftrightarrow \forall x. x \in A \Rightarrow x \in B \tag{5.23}$$

$$\forall A, B, x. x \in A \cup B \Leftrightarrow x \in A \vee x \in B \tag{5.24}$$

The textbook proof looks as follows:

Textbook Proof:

"$\subset$": We show $A \cup B \subset B \cup A$

1. Assume $x \in A \cup B$
2. Thus $x \in A \vee x \in B$

   Case $x \in A$:

   (a) It follows that $x \in B \vee A$

   (b) It follows that $x \in B \cup A$

   Case $x \in B$:

   (a) It follows that $x \in B \vee A$

   (b) It follows that $x \in B \cup A$

"$\supset$": analogously.

Within Core, the initial proof state is given by

$$(5.22) \wedge (5.23) \wedge (5.24) \Rightarrow (5.21) \tag{5.25}$$

Instead of considering the proof state as a whole, we identify (5.22) ... (5.24) to represent the available assertions, and (5.21) as the conjecture to be shown. Before starting the actual proof attempt, we preprocess the assumptions to make subsequent resolution replacement rules possible. Alternatively, it is possible to apply rewrite replacement rules directly. However, we refrain from doing so, as rewrite replacement rules are mapped back to resolution replacement rules.

The preprocessing consists of isolating one direction of the equivalence, i.e., going from $(A \Leftrightarrow B)^-$ to $(A \Rightarrow B)^-$ by the following sequence of CORE calculus rules:

1. apply Leibniz equality to $(A \Leftrightarrow B)^-$ to obtain $(\forall P.P(A) \Rightarrow P(B))^-$ (as well as a copy of $(A \Leftrightarrow B)^-$ in α-relation)
2. instantiate P by $\lambda x.x$, resulting in $(A \Rightarrow B)^-$ via β-reduction.
3. weaken the parent of the node with label $(A \Rightarrow B)^-$ to remove the copy

The process is illustrated in detail for the assertion (5.22) in Figure 5.2[3].
Similarly, the other assertions are preprocessed, resulting in the following assertions:

$$\forall A, B. \left((A \subset B)^+ \wedge^\beta (B \subset A)^+\right)^+ \Rightarrow^\beta (A = B)^-\Big)^- \tag{5.26}$$

$$\forall A, B.\forall x. \Big((x \in A)^- \Rightarrow (x \in B)^+ \Rightarrow^\beta (A \subset B)^-\Big)^- \tag{5.27}$$

$$\forall A, B.\forall x. \Big(((x \in A)^+ \vee^\alpha (x \in B)^+)^+ \Rightarrow^\beta (x \in A \cup B)^-\Big)^- \tag{5.28}$$

Note that for all assertions, the other direction of the equivalence can be obtained in a similar way; however, for the simple proof above we only need

$$(\forall A, B.\forall x.(x \in A \cup B)^+ \Rightarrow^\beta (x \in A \vee^\beta x \in B)^-)^- \tag{5.29}$$

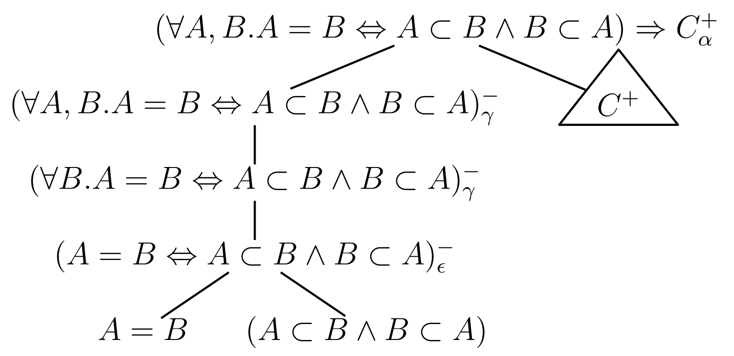
The structure of the resulting formula tree is

$$(5.26) \wedge (5.27) \wedge (5.24) \Rightarrow (5.21) \tag{5.30}$$

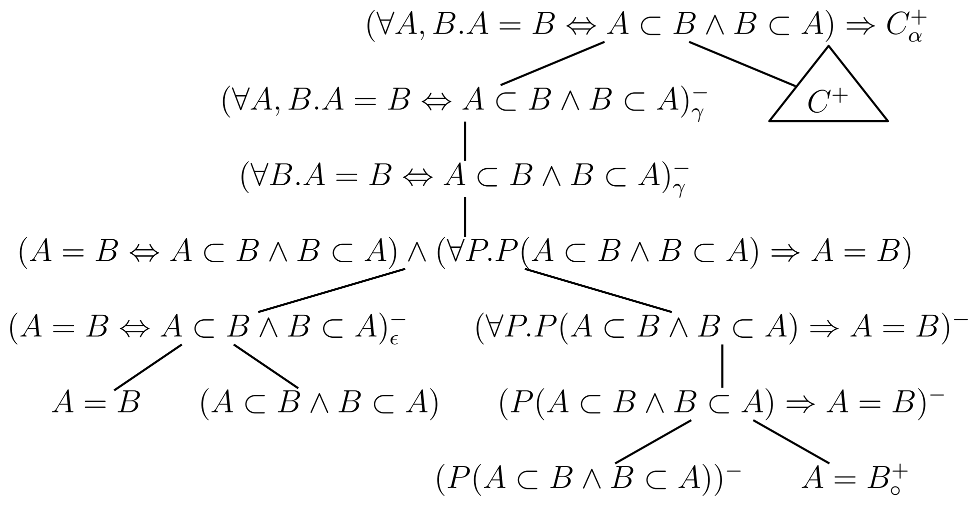
The result of preprocessing the assertions is that more subformulas have polarities and can therefore be used for transformations via resolution replacement rules. However, as a single assertion may be needed several times, as for example the definition of $\cup$ in the example proof above, we will need a second operation to copy that assertion. Fortunately, we can make use of CORE's rule to increase the multiplicity; in our case this usually results in an α-insertion of a copy of the assertion, as assertions are usually universally quantified at top-level. In principle, increasing of the multiplicity of an assertion works as follows:

- Let Q be the subtree of the indexed formula tree containing the assertion. Moreover, let $\mathcal{Q}$ be the set of all nodes within Q that are minimal with respect to the structural ordering $\prec_Q$ and are of secondary type γ_0.
- For each $Q' \in \mathcal{Q}$ increase the multiplicity of Q'. Note that the only substitutions stemming from the application of the Leibniz equality might be present. As these substitutions do not involve δ-variables, $\mathit{Inst}_Q(Q'') = \emptyset$ for all nodes Q'' within Q, and therefore $\mu(Q') = \{Q'\}$ for all $Q' \in \mathcal{Q}$. Moreover, all $Q' \in \mathcal{Q}$ are therefore convex.

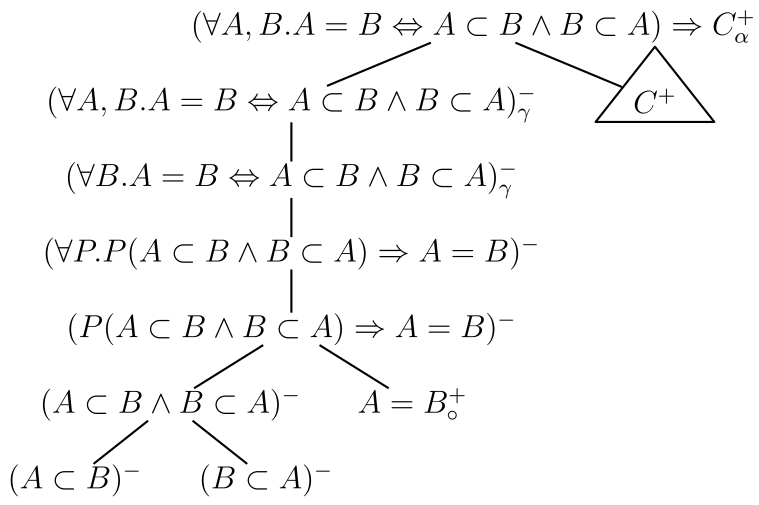
Again, we illustrate the operation using the assertion (5.22). Figure 5.3 shows the indexed formula tree before and after the increase of the multiplicities. The minimal γ-node that needs to be copied corresponds to the $\forall A$ quantifier, and the corresponding subtree contains the complete assertion. The subtree is copied, the variables renamed and


$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \Rightarrow C^+_\alpha$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
C^+
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\epsilon$
$A = B$
$(A \subset B \wedge B \subset A)$

(a) Indexed formula tree before the application of Leibniz' equality

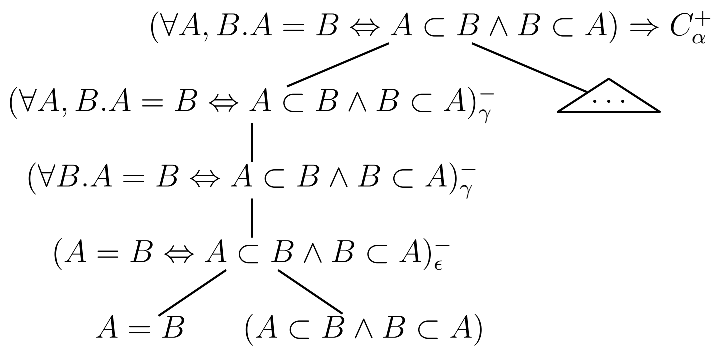

$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \Rightarrow C^+_\alpha$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
C^+
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A) \wedge (\forall P.P(A \subset B \wedge B \subset A) \Rightarrow A = B)$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\epsilon$
$(\forall P.P(A \subset B \wedge B \subset A) \Rightarrow A = B)^-$
$A = B$
$(A \subset B \wedge B \subset A)$
$(P(A \subset B \wedge B \subset A) \Rightarrow A = B)^-$
$(P(A \subset B \wedge B \subset A))^-$
$A = B^+_\circ$

(b) Indexed formula tree after the application of Leibniz' equality, before weakening

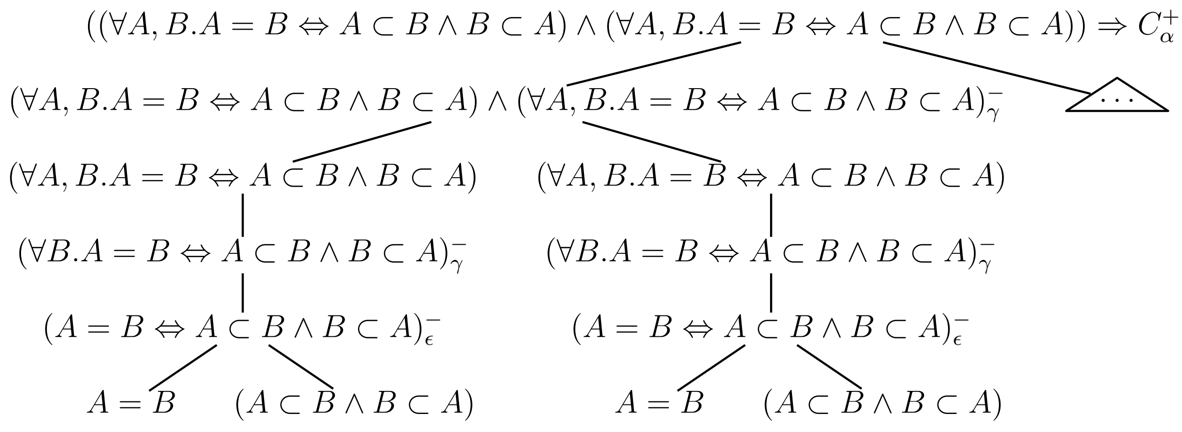

$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \Rightarrow C^+_\alpha$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
C^+
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)^-_\gamma$
$(\forall P.P(A \subset B \wedge B \subset A) \Rightarrow A = B)^-$
$(P(A \subset B \wedge B \subset A) \Rightarrow A = B)^-$
$(A \subset B \wedge B \subset A)^-$
$A = B^+_\circ$
$(A \subset B)^-$
$(B \subset A)^-$

(c) Indexed formula tree after weakening and instantiation

Figure 5.2: Preprocessing of the assertion (5.22)


$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \Rightarrow C_\alpha^+$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)_\gamma^-$
$\cdots$
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)_\gamma^-$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A)_\epsilon^-$
$A = B$
$(A \subset B \wedge B \subset A)$

(a) Indexed formula tree before increasing the multiplicity


$((\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \wedge (\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)) \Rightarrow C_\alpha^+$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A) \wedge (\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)_\gamma^-$
$\cdots$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)$
$(\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A)$
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)_\gamma^-$
$(\forall B.A = B \Leftrightarrow A \subset B \wedge B \subset A)_\gamma^-$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A)_\epsilon^-$
$(A = B \Leftrightarrow A \subset B \wedge B \subset A)_\epsilon^-$
$A = B$
$(A \subset B \wedge B \subset A)$
$A = B$
$(A \subset B \wedge B \subset A)$

(b) Indexed formula tree after increasing the multiplicity

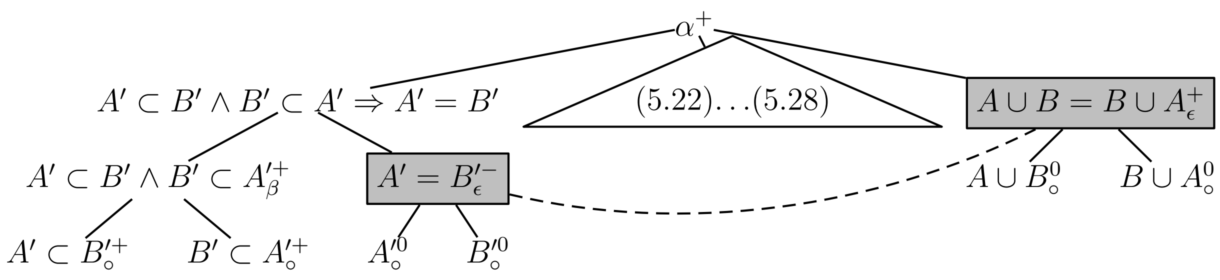
Figure 5.3: Increasing the multiplicity of the assertion (5.22)

the substitution $P \mapsto \lambda x.x$ propagated. Therefore, our previous preprocessing is not lost and a new copy of the preprocessed assertion available.

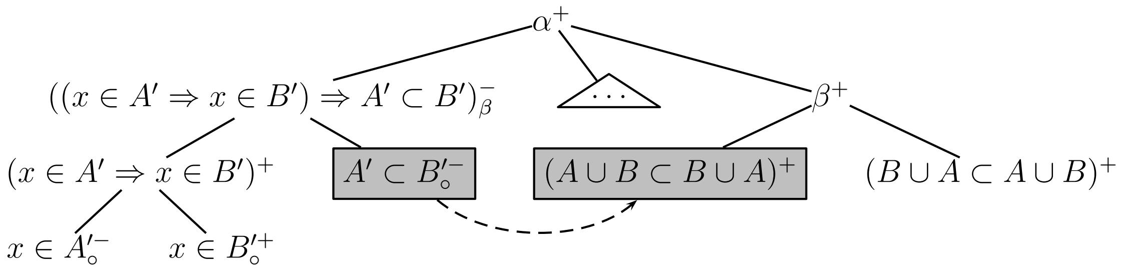
Now we are prepared to show the first direction of the proof, relying only on resolution replacement rule applications of a special form.

Formal Proof:

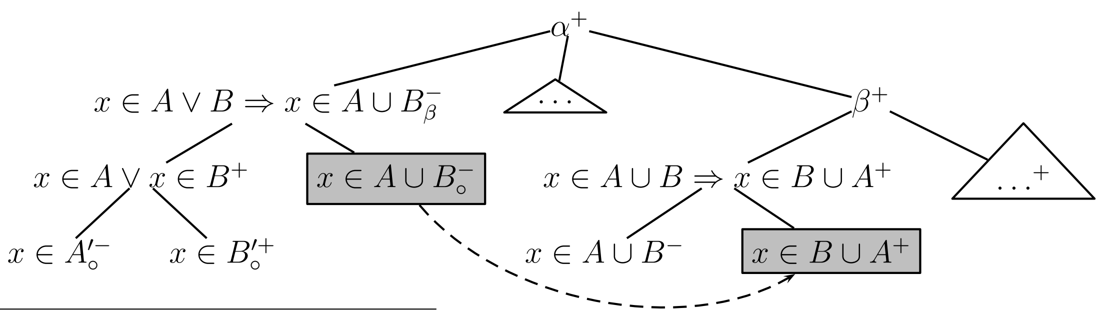
After preprocessing and copying the assertion (5.26) which will be used in the first proof step, the free variable indexed formula tree looks schematically as follows:



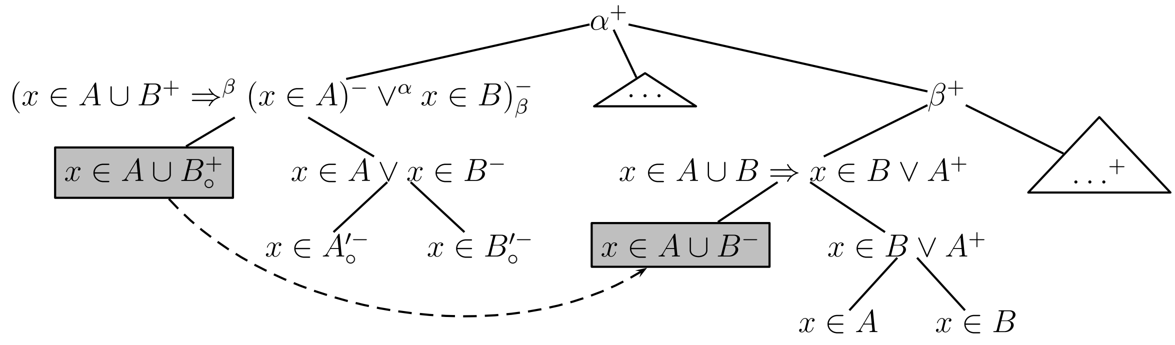
We observe that $A' = B'^-$ and $(A \cup B = B \cup A)^+$ have opposite polarity and are connectable via the substitution $A' \mapsto A \cup B, B' \mapsto B \cup A$. Applying the substitution and then a resolution replacement rule $A \cup B = B \cup A \to \langle A \cup B \subset B \cup A^+, (B \cup A \subset A)^+ \rangle$, before invoking the simplification rule results in the following free variable indexed formula tree:

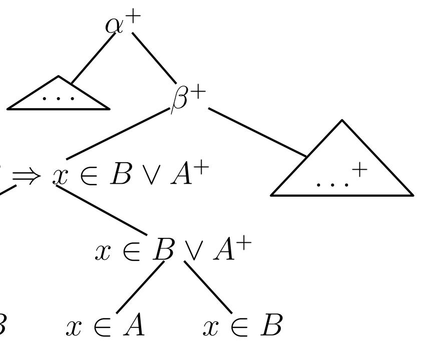

α^+
$((x \in A' \Rightarrow x \in B') \Rightarrow A' \subset B')^-_\beta$
$\ldots$
β^+
$(x \in A' \Rightarrow x \in B')^+$
$A' \subset B'^-_\circ$
$(A \cup B \subset B \cup A)^+$
$(B \cup A \subset A \cup B)^+$
$x \in A'^-_\circ$
$x \in B'^+_\circ$

As before, we apply the substitution $A' \mapsto A \cup B, B' \mapsto B \cup A$ and then a resolution replacement rule $A \cup B \subset B \cup A \to \langle x \in A \cup B \Rightarrow x \in B \cup A^+ \rangle$, apply the simplification rule and preparing the next assertion results in the following free variable indexed formula tree.



[3]We only show the indexed formula tree, as the free variable indexed formula tree can easily be obtained from it

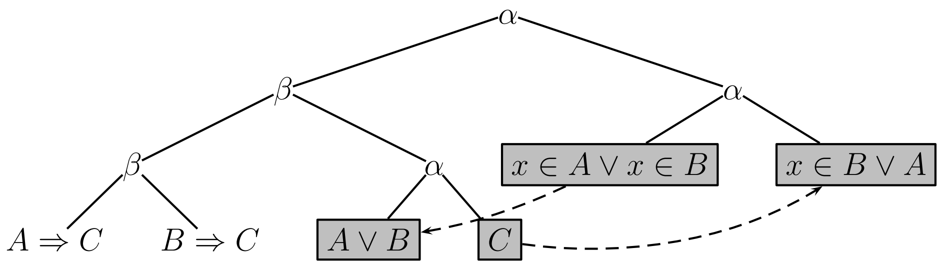




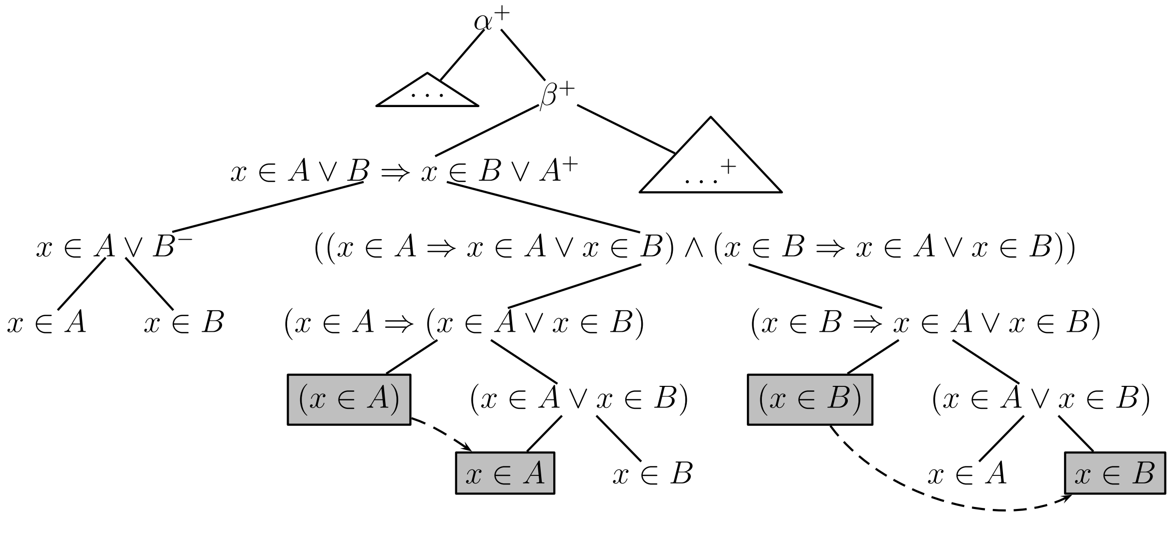
The case distinction of the textbook proof can either be modeled using Schütte's β-decomposition rule, or we can use of the following lemma, which is provable in Core:

$$\forall A'_o, B'_o, C'_o.((A' \Rightarrow C')^+ \wedge^\beta (B' \Rightarrow C')^+) \Rightarrow^\beta ((A' \vee B') \Rightarrow C')^- \tag{5.31}$$

We take the second option, because it shows more clearly how Core can be used as a meta-framework. The lemma together with our current subgoal has the following structure:



Application of the lemma results in the following free variable indexed formula tree, in which the first subgoal can trivially be closed:



α^+
β^+
$x \in A \vee B \Rightarrow x \in B \vee A^+$
$\ldots^+$
$x \in A \vee B^-$
$((x \in A \Rightarrow x \in A \vee x \in B) \wedge (x \in B \Rightarrow x \in A \vee x \in B))$
$x \in A$
$x \in B$
$(x \in A \Rightarrow (x \in A \vee x \in B)$
$(x \in B \Rightarrow x \in A \vee x \in B)$
$(x \in A)$
$(x \in A \vee x \in B)$
$(x \in B)$
$(x \in A \vee x \in B)$
$x \in A$
$x \in B$
$x \in A$
$x \in B$

5.3.2 Equational Reasoning

We now present a proof of the theorem about sums of natural numbers: $\forall n \sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$. The proof works by induction and relies on equational reasoning.

Textbook Proof (from [Aut03] p. 96):

of $\forall n \sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$**.** The proof is by induction over n:

Base Case $n = 0$**:**

1. We have to show $\sum_{i=1}^{0} i^3 = (\sum_{i=1}^{0} i)^2$.
2. By definition of $\sum$ we obtain $0 = 0$

Induction Step $n \to n + 1$**:** The induction hypothesis is $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$.

1. We have $\sum_{i=1}^{n+1} i^3 = (\sum_{i=1}^{n+1} i)^2$.
2. By definition of $\sum$ we obtain $(n + 1)^3 + \sum_{i=1}^{n} i^3 = ((n + 1) + \sum_{i=1}^{n} i)^2$.
3. By $(a + b)^2 = a^2 + 2ab + b^2$ we obtain $(n + 1)^3 + \sum_{i=1}^{n} i^3 = (n + 1)^2 + 2(n + 1)(\sum_{i=1}^{n} i) + (\sum_{i=1}^{n} i)^2$.
4. By Ind. Hyp. it reduces to $(n + 1)^3 = (n + 1)^2 + 2(n + 1)(\sum_{i=1}^{n} i)$.
5. By $\sum_{i=1}^{n} = \frac{n(n+1)}{2}$ we obtain $(n + 1)^3 = (n + 1)^2 + 2(n + 1)\frac{n(n+1)}{2}$
6. And finally $(n + 1)^3 = (n + 1)^3$ □

Formal Proof

For the formal proof we assume the definition of the operator sum as used in Example 5.2.15. Moreover, we assume the following assertions to keep the proof short:

$$\forall p.(p(0) \land \forall y.p(y) \Rightarrow p(s(y))) \Rightarrow \forall x.p(x) \tag{5.32}$$

$$\forall n, m. \sum_{i=1}^{s(m)} i^n = s(m)^n + \sum_{i=1}^{m} i^n \tag{5.33}$$

$$\forall a, b.(a + b)^2 = a^2 + 2ba + b^2 \tag{5.34}$$

$$\forall a, b, c.a = b \Rightarrow a + c = b + c \tag{5.35}$$

$$\forall n. \sum_{i=1}^{n} i^1 = \frac{ns(n)}{2} \tag{5.36}$$

$$\forall m.2\frac{m}{2} = m \tag{5.37}$$

$$\forall q.s(q)^3 = s(q)^2 + (q(s(q)s(q))) \tag{5.38}$$

$$0 = 0^2 \tag{5.39}$$

To keep the proof readable, we only show the part of the proof tree which corresponds to the goal formula. Note that the complete formula tree is much larger. In particular, it contains each of the assertions. Moreover, the assertions need to be copied for each assertion application. The proof is a slight modification of the proof given in [Aut03] p. 96.

In the first step, we apply the induction axiom for natural numbers and obtain

$$\sum_{i=1}^{0} i^3 = \left(\sum_{i=1}^{0} i\right)^2 \wedge \forall n. \wedge \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow \sum_{i=1}^{s(n)} i^3 = \left(\sum_{i=1}^{s(n)} i\right)^2 \tag{5.40}$$

We apply twice the definition of $\sum$ which results in

$$0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow \sum_{i=1}^{s(n)} i^3 = \left(\sum_{i=1}^{s(n)} i\right)^2 \tag{5.41}$$

Applying the definition of $\sum$ twice results in

$$0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow (s(n))^3 + \sum_{i=1}^{n} i^3 = \left(s(n) + \sum_{i=1}^{n} i\right)^2 \tag{5.42}$$

By $(a+b)^2 = a^2 + 2ab + b^2$ we obtain

$$0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow (s(n))^3 + \sum_{i=1}^{n} i^3 = s(n)^2 + 2s(n)\left(\sum_{i=1}^{n} i\right) + \left(\sum_{i=1}^{n} i\right)^2 \tag{5.43}$$

Applying the induction hypothesis $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$ and subsequent simplification by $(a+b = c+b) \Leftrightarrow (a = c)$ we obtain

$$0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow (s(n))^3 = (s(n))^2 + 2(n+1)\left(\sum_{i=1}^{n} i\right) \tag{5.44}$$

Applying $\sum_{i=1}^{n} = \frac{n(n+1)}{2}$ to $\sum_{i=1}^{n} i$ results in

$$\forall n. n = 0 \Rightarrow 0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow (s(n))^3 = (s(n))^2 + 2(s(n))(\frac{ns(n)}{2}) \tag{5.45}$$

which after some further simple rearrangements results in

$$0 = 0 \wedge \forall n. \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 \Rightarrow s(n)^3 = s(n)^3 \tag{5.46}$$

A subsequent simple simplification shows that now the proof is completed.

5.4 Summary

In this chapter we introduced Autexier's CORE calculus on which our work will be based. Thereby, we instantiated CORE's meta theory to higher-order logic to ease the technical presentation considerably. We gave two example proofs to illustrate the style of reasoning supported by CORE and highlighted the differences between informal presentation of these proofs and the corresponding formal versions in CORE's proof theory. Moreover, we sketched how a CORE proof state can be organized to ease the application of assertions by dividing the proof state into two parts: one part corresponding to the assertions and one part corresponding to the conjecture to be shown. We will follow this approach in the next chapter. Finally, we illustrated how an assertion application can be mapped to a sequence of CORE calculus rule applications.

6 The CORE calculus and the Assertion Level

Almost all interactive theorem provers are based on the sequent calculus or natural deduction. This is because proofs in these calculi are considered to be more human oriented and are therefore better suited for proof communication between man and machine. In particular, they provide the intuitive notion of a subgoal which allows for a convenient presentation of the state of the current proof process. In contrast, machine oriented calculi, such as resolution (see for example [Gab00] p. 2 for a discussion), are designed for efficiency, at the cost that the resulting proofs are very different in their structure in comparison to human proofs. Moreover, given a partial derivation, it is even more difficult for a mathematician to get a sense of the current state of the proof: is a proof state still provable, is it difficult to solve it in principle, or how many steps are approximately needed to close the goal.

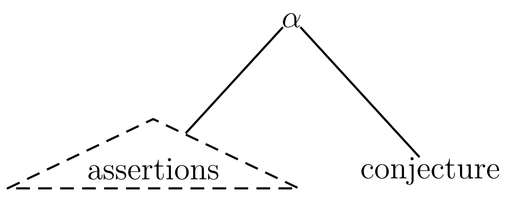
However, compared to other calculi, the degree of automation is rather low in the sequent calculus. One reason for this is that the sequent calculus has to follow the logical structure of the formula. In particular, formulas have to be decomposed in order to get access to intermediate subformulas, with the consequence that the resulting proof trees are redundant to some degree. In contrast, matrix based calculi do not require the decomposition of formulas, but rely on the notion of a connection, resulting in a compact representation of the proof state and very efficient search procedures. However, this comes at the cost that proofs are no longer readable (see e.g. [ABI$^+$96] p. 325). Moreover, the definition of search strategies/tactics on top of matrix methods is very difficult.

CORE's proof theory, which has been introduced in the previous chapter, can be seen as a hybrid approach: The initial free variable indexed formula tree can be understood as a matrix representation of the original conjecture. However, instead of searching for a complete set of connections, the free variable indexed formula tree is transformed with the goal to produce the trivially valid subgoal true by sound and complete contextual reasoning. CORE provides uniform notation to determine the context of an arbitrary subformula statically and to modify subformulas by the application of replacement rules. Therefore, there is no need to follow the logical connectives and to decompose formulas, as needed in the sequent calculus. However, there are the following disadvantages: (i)

The complete theory as well as the goal formula need to be encoded within a single indexed formula tree, which, as a consequence, can become very large. Presenting an unstructured indexed formula tree in the presence of large theories is therefore not an option. (ii) Replacements in an unstructured indexed formula tree are always global; therefore the computation of the effect of a replacement is expensive. (iii) The need to talk about positions and their interrelation makes it difficult to define proof strategies on top of the calculus.

In this section, we define an assertion level interface on top of the Core calculus, combining the features of Core and the sequent calculus. The interface hides the internal representation of the proof state as an indexed formula tree and provides a sequent-style interface instead. The interface is based on sequents and inferences to transform these sequents. As in the sequent calculus, search procedures can now conveniently be defined on top of inferences. However, in contrast to the classical sequent calculus, the number of inferences is not static, but grows with the number of theorems that are proven.

The main idea is to divide the indexed formula tree into two parts: a theory part, which contains the available assertions, and a conjecture part, containing the conjecture the user attempts to prove. However, only the latter is presented to the user, as illustrated in the diagram below:



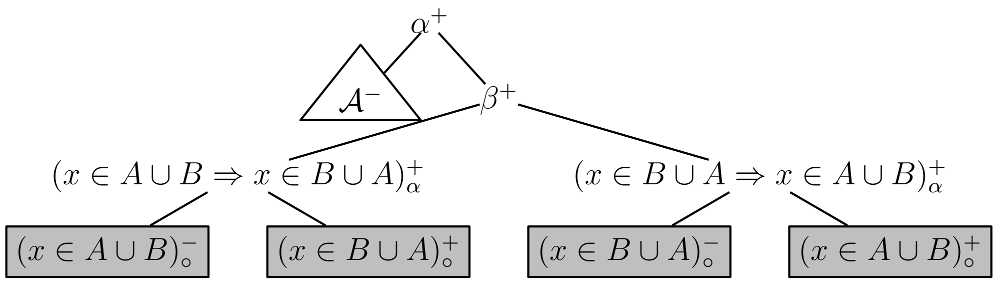
This reduces the size of the (shown) proof state dramatically. In addition, we also obtain computational benefits by imposing a special structure on the indexed formula tree. Deciding whether an assertion is applicable with respect to the current (local) proof state of the conjecture becomes a local property: The assertion itself and the subgoal to which it is applied are sufficient to compute the modification of the subgoal. Moreover, the multiplicities for assertions are automatically handled by the assertion level interface.

6.1 Windows and Inference Representation

To be able to represent a subgoal in the sequent style as a single formula together with a list of its assumptions, we follow the idea of window inference (see [RS93]) and introduce the notion of a task as a set of windows. Window inference was originally introduced as a mechanism to focus on specific parts of a formula by transforming the formula relative to its context and finally unfocusing the part of the formula. Window inferencing in the same style has been defined on top of the Core calculus in [Aut03], where the main advantage is that windows do not introduce proof obligations that have to be discharged.

Before introducing the terminology of a window, we illustrate the concept by an example.

Example 6.1.1. *Let us reconsider the proof given in Section 5.3 of the conjecture $A \cup B \subset B \cup A$ after the application of set extensionality and the definition of subset. The resulting indexed formula tree is shown below.*


α^+
$\mathcal{A}^-$
β^+
$(x \in A \cup B \Rightarrow x \in B \cup A)^+_\alpha$
$(x \in B \cup A \Rightarrow x \in A \cup B)^+_\alpha$
$(x \in A \cup B)^-_\circ$
$(x \in B \cup A)^+_\circ$
$(x \in B \cup A)^-_\circ$
$(x \in A \cup B)^+_\circ$

The part of the indexed formula tree containing the assertions is shown on the left and folded in the subtree $\mathcal{A}$, and on the right we see the current proof state. A representation in the style of the sequent calculus can be obtained by marking special nodes of the formula tree by windows, which are indicated by boxes above. Provided that all parts of a subtree R are covered by a given set of windows – we will call such a set spanning *with respect to R – it is possible to use this set to represent the proof state corresponding to R in a convenient form: A sequent is given by a set of α-related windows, where a window with negative polarity corresponds to a formula in the antecedent and a window with positive polarity to a formula in the succedent of the sequent. β-nodes above windows introduce multiple sequents. For our example, we get the following two sequents*

$$x \in A \cup B \vdash x \in B \cup A \tag{6.1}$$

$$x \in B \cup A \vdash x \in A \cup B \tag{6.2}$$

Putting the subgoals together, we obtain a so-called agenda, which is a set of sequents together with an overall substitution σ.

$$\langle (6.1), (6.2); \{\} \rangle \tag{6.3}$$

In the sequel we define the notion of windows for subtrees of free variable indexed formula trees, which may denote any subtree of R. The target domain of some window for R is defined as follows:

Definition 6.1.2 (Substructures of free variable indexed formula trees)**.** *Let R be a free variable indexed formula tree. The* substructures *of R are all subtrees of R. We denote that set by $\mathcal{S}(R)$. Those $S \in \mathcal{S}(R)$ that are leaf nodes annotated by some π are called* inner substructures.

Notation 6.1.3. *As usual, each node in the tree can be uniquely identified by a path from the root node to this node. We represent such a path by a sequence of natural numbers and use ϵ to refer to the root node of the tree.*

Based on substructures, we model all windows of a formula tree as a partial mapping from identifiers to such substructures.

Definition 6.1.4 (Windows)**.** *Let R be a free variable indexed formula tree, $\mathcal{W}$ an enumerable set, and $f : \mathcal{W} \hookrightarrow \mathcal{S}(R)$ a partial function. We say that f is a* window structure *for R and each $n \in dom(f)$ is a* window *that denotes the subtree $f(n)$. The polarity, uniform type and label of n are those of $f(n)$, if $f(n)$ is a subtree. Otherwise $f(n) := R_\pi$ and n has undefined polarity $(\cdot)$ and uniform type $(\circ)$, and its label is $\textbf{\textit{Label}}(R)_{|\pi}$. The* windows in some substructure S with respect to f, *denoted by* Win(S, f) *are all $n \in dom(f)$ such that $f(n) \in \mathcal{S}(S)$.*

We denote by (S, f) the combination of a substructure with a window structure f for S, and say that S is annotated by *f. We say that f is spanning for S iff any literal node of S is contained in a subtree denoted by one of the windows of f.*

Example 6.1.5. *Consider the indexed formula tree of example 6.1.1, in which windows are depicted by boxes. The window structure is spanning for the task part, i.e., the subtree rooted at the β^+ node, but is not spanning for the complete indexed formula tree, as there are literal nodes L inside $\mathcal{A}$ with* $\mathrm{Win}(L, f) = \emptyset$.

While proving a conjecture, it will be necessary to replace some substructure of the indexed formula tree by another substructure. To be able to transfer annotations from the original tree to the tree resulting from the replacement, we identify corresponding substructures of both trees. Indeed, given a substructure S' of S which is to be replaced by a substructure S'', it is possible to define a mapping ι that identifies the corresponding parts which are not affected by the replacement. It is defined as follows:

Definition 6.1.6 (Replacement of Substructures). *Let S, S' be substructures of some free variable indexed formula tree, $S' \in \mathcal{S}(S)$, and S'' a substructure of another free variable indexed formula tree. Then we denote by $(S_{|S' \leftarrow S''}, \iota)$ the replacement of S' with S'' in S together with a partial mapping $\iota : \mathcal{S}(S) \setminus \mathcal{S}(S') \hookrightarrow \mathcal{S}(S_{|S' \leftarrow S''})$ which is defined by*

- *If S' is a subtree and S'' is a subtree then it denotes the standard replacement of S' with S''; ι is the mapping of the substructures of S not in S' to their corresponding substructures in $S_{|S' \leftarrow S''}$.*
- *If $S' := R'_\pi$ and $S'' := R''_\pi$, and if the labels of R' and R'' are equal up to the subterms denoted by π, then $S_{|S' \leftarrow S''}$ denotes the replacement of R' by R''; ι is the mapping of the substructures of S that are not in S' to their corresponding substructures in $S_{|S' \leftarrow S''}$.*
- *Otherwise the replacement is undefined.*

Similar to the case above, we can replace a substructure S' of an annotated substructure (S, f) by an annotated substructure (S'', f'), thereby using ι as defined above to transfer the windows into the resulting formula tree:

Definition 6.1.7 (Replacement of Annotated Substructures). *Let $(S, f), (S'', f')$ be annotated substructures, for S, and S' a substructure of S. The* replacement $(S, f)_{|S' \leftarrow (S'', f')}$ of S' by (S'', f') in (S, f) *is defined iff $(S_{|S' \leftarrow S''}, \iota)$ is defined and $(dom(f) \setminus \mathrm{Win}(S', f)) \cap dom(f') = \emptyset$ holds. If it is defined, the replacement results in (S^*, f^*) where $S^* := S_{|S' \leftarrow S''}$ and f^* is defined by*

- *If either $S' \neq S$, or $S' := S$ then*

$$f^*(n) := \begin{cases} \iota(f(n)) & \textit{if } n \in dom(f) \setminus \mathrm{Win}(S', f) \\ f'(n) & \textit{if } n \in dom(f') \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

- *Otherwise, if $S' = S$ and there is no $n \in dom(f')$ with $f'(n) = S''$, then assume $n_0 \notin dom(f')$ in*

$$f^*(n) := \begin{cases} f'(n) & \textit{if } n \in dom(f') \\ S'' & \textit{if } n = n_0 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Following Example 6.1.1, we now use windows to define the notion of a sequent. To be a sequent, a set of windows must satisfy the property that the windows cover the complete subtree, and the individual windows must be α-related to each other. We thus arrive at the following definition:

Definition 6.1.8 (Sequent)**.** *Let R be a free variable indexed formula tree, f be a window structure for R. Moreover, let $w_1, \dots, w_n$ windows, such that $w_1, \dots, w_i$ $(1 \leq i \leq n)$ have negative polarity and $w_{i+1}, \dots, w_n$ have positive polarity. Let further R' be the smallest subtree that contains all subtrees denoted by $w_1, \dots, w_n$. Then $w_1, \dots, w_i \vdash w_{i+1}, \dots, w_n$ is a* sequent with respect to R' *iff*

1. *all w_i are α-related between each other, and*

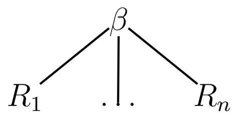
2. *there is no subtree in R' that is β-related to any w_i.*

We say that a sequent $w_1, \dots, w_i \vdash w_{i+1}, \dots, w_n$ is proved *iff at least one of the w_i denotes a subtree that is proved, i.e., is either* true$^+$, false$^-$*, or $\zeta(s, s)$.*

Notation 6.1.9. *In the sequel, we agree to denote a sequent $w_1, \dots, w_i \vdash w_{i+1}, \dots, w_n$ also by simply writing the list of window $w_1, \dots, w_i, w_{i+1}, \dots, w_n$, since the sequent-structure is uniquely determined up to permutations of windows by the polarities of the windows. Furthermore, we may write $\varphi_1^{p_1}, \dots, \varphi_n^{p_n}$ to denote a sequent composed of n windows, each denoting a subtree of label φ_i and polarity p_i.*

It is natural to generalize the notion of a single sequent to a set of sequents to be able to express splittings as in the sequent calculus.

Definition 6.1.10 (Sequential)**.** *Let $S_1, \dots, S_n$ be a set of sequents. For each i $(1 \leq i \leq n)$ let R_i denote the smallest subtree that contains all windows corresponding to the sequent S_i. We say that $S_1, \dots, S_n$ are sequential iff the R_i are β-related among each other, as shown in the picture below:*



We are now in the position to define the notion of indexed task trees and their free variable counterpart. Indexed task trees (respectively free variable indexed task trees) are indexed formula trees (respectively free variable indexed formula trees) that have the specific form $\alpha(\mathcal{A}, G)$, where $\mathcal{A}$ is a set of assertions and G is a formula to be proved, together with a window structure which identifies sequents in the subtree corresponding to G. As for indexed formula trees, we define task trees in two steps: First, we define the task tree obtained from a set of assertions $\mathcal{A}$ and a conjecture G, which we call initial indexed task tree. We subsequently define operations to transform both trees.

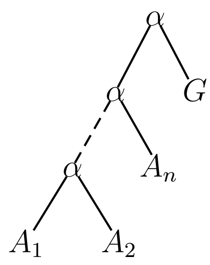
Definition 6.1.11 (Initial Indexed Task Tree, Free Variable Indexed Task Tree, Task Window Proof State)**.** *Let $\mathcal{A}$ be a set of assertions and let G be a conjecture to be proved. Then the* initial indexed task tree *for $\mathcal{A}$ and G is the indexed formula tree of the formula*

$$\left[\bigwedge_{A\in\mathcal{A}} A \Rightarrow \boxed{G}\right] \tag{6.4}$$

The initial free variable indexed task tree *is the initial free variable indexed formula tree of indexed task tree.*

Due to its initialization, the free variable indexed task tree obeys the following structural property:

Theorem 6.1.12 (Structural Property of the Initial Indexed Task Tree)**.** *Let $\mathcal{A}$ be a set of assertions and let G be a conjecture to be proved. The initial indexed task tree for $\mathcal{A}$ and G has the following form:*



In particular, for any $A \in \mathcal{A}$, there are no β-related formulas on the path from the node labeled with A to the root node of the free variable indexed task tree.

Proof. By induction on $n = |\mathcal{A}|$. □

Notation 6.1.13. *We denote the (free variable) indexed task tree for assertions $\mathcal{A}$ and a goal G by $\alpha(\mathcal{A}, G)$. Moreover, we do not differentiate between a node labeled with a formula A and the formula itself if the context is clear.*

Corollary 6.1.14 (Locality of β-related nodes)**.** *Let $\alpha(\mathcal{A}, G)$ be an initial free variable indexed task tree and $A \in \mathcal{A}$ be an assertion. Moreover, let $n \in \mathcal{S}(A)$, and let $n' \in \mathcal{S}(\alpha(\mathcal{A}, G))$. If n' is β-related to n, then $n' \in \mathcal{S}(A)$.*

Proof. If n' is not in $\mathcal{S}(A)$, then the least node that governs both n and n' must be of type α due to Theorem 6.1.12. □

We now introduce the notion of a task proof state on top of the concepts defined so far. The overall goal is to arrive at the intuitive notion of an agenda, which contains exactly all proof obligations that need to be discharged to close the overall conjecture, but completely hides the underlying concepts of the formula trees. That is, from a user's perspective, a proof state looks like a proof state in the sequent calculus. However, as we will see later, the transformations provided by the Core calculus are much more general and will allow the construction of shorter proofs.

Definition 6.1.15 ((Initial) Task Proof State)**.** *Let $\mathcal{A}$ be a set of assertions and let G be a conjecture to be proved. The* task proof state *consists of the following components:*

- *the indexed task tree Q=$\alpha(\mathcal{A}, G)$*
- *the corresponding free variable representation of $\alpha(\mathcal{A}, G)$*
- *a substitution σ*
- *a window structure f for G with a set of sequents $S_1, \dots, S_n$.*

It is denoted by $[Q; \sigma; (\alpha(\mathcal{A}, G), f) \rhd \{S_1, \dots, S_n\}]$. The initial task proof state *is given by the initial indexed task tree, the initial free variable indexed task tree, the empty substitution, and the window structure $f : \mathcal{W} \hookrightarrow \mathcal{S}(\alpha(\mathcal{A}, G))$ with $dom(f) = \{c\}$ and*

$$f(n) := \begin{cases} G & n = c \\ \mathit{undefined} & \mathit{otherwise} \end{cases} \tag{6.5}$$

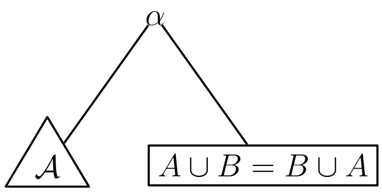
To emphasize the difference to the sequent calculus we call the sequents tasks. $[Q; \sigma; (\alpha(\mathcal{A}, G), f) \rhd \{S_1, \ldots, S_n\}]$ *is consistent, if f consists of a set of sequents that are spanning for G and the sequents are sequential. A task proof state is* proved *iff all tasks of that task proof state are proved.*

Hiding the task tree and its free variable counterpart, we arrive at the definition of an agenda.

Definition 6.1.16. *Let $Q, \sigma \rhd (\alpha(\mathcal{A}, G), f)$ be a consistent task proof state with tasks $T_1, \ldots, T_n$, one of which is marked by $\underline{\cdot}$ and called* current task. *The list of tasks together with the substitution σ is called* agenda *and denoted by*

$$\langle \underline{T_1}, \ldots, T_n; \sigma \rangle \tag{6.6}$$

Example 6.1.17. *For the first example from Section 5.3, the initial proof state is*



The initial task is $A \cup B = B \cup A$. The initial agenda is

$$\langle \underline{A \cup B = B \cup A}; \{\} \rangle \tag{6.7}$$

Let us stress here the close connection between a task proof state $[Q; \sigma; R \rhd \{S_1, \ldots, S_n\}]$ and the corresponding Core proof state $[Q, \sigma \rhd R]$ (where $R = \alpha(\mathcal{A}, G)$). In particular, any transformation

$$[Q, \sigma \rhd R] \mapsto [Q', \sigma' \rhd R'] \tag{6.8}$$

can be mapped to a transformation

$$[Q; \sigma; (R, f) \rhd \{S_1, \ldots, S_n\}] \mapsto [Q'; \sigma; (R', f') \rhd \{S'_1, \ldots, S'_m\}] \tag{6.9}$$

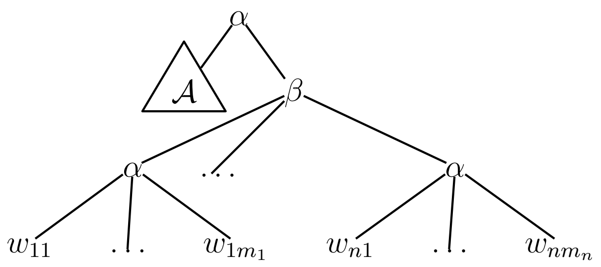
provided that the window structure f is adapted accordingly, and vice versa.

Theorem 6.1.18 (Consistency of initial task proof state)**.** *The initial task proof state is consistent.*

Proof. There is only one window w_1^+ with positive polarity which points to the initial conjecture G. Therefore, all conditions in 6.1.8 are trivially satisfied; moreover, w_1 is spanning for G. □

Theorem 6.1.19 (Accuracy of the Definition)**.** *Definition 6.1.15 is accurate.*

Proof. The intuition in the above definition is that a task proof state is proven if the underlying Core proof state can be shown. This is indeed the case: A proven task proof state is of the form $[Q; \sigma; (R, f) \rhd \{S_1, \ldots, S_n\}]$, where $S_i = w_{i1}, \ldots, w_{in_i}$ such that some w_{ij_1} is $True^+, False^-$, or $\zeta(s, s)$.



Consequently, we have proved the subtree corresponding to w_i. Therefore, applying the simplification rule (see Definition 5.2.19) to the root of the free variable indexed formula tree results in the free variable indexed formula tree $true^+$. $\Box$

6.2 Representing Assertions

Given an initial task proof state $[Q;\sigma;(R,f) \rhd \{S_1,\dots,S_n\}]$, we are now concerned with defining transformations on it. These transformations are of the form

$$[Q;\sigma;(R,f) \rhd \{S_1,\dots,S_n\}] \mapsto [Q';\sigma;(R',f') \rhd \{S'_1,\dots,S'_m,S_2,\dots,S_n\}] \tag{6.10}$$

and will consist of assertion applications. Instead of representing assertions as individual formulas, we will present them in the computational form of inference rules. While the standard form of inferences in the sequent calculus is like

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \tag{6.11}$$

this representation is no longer intuitive when relaxing the condition that inferences can only be applied at top-level. Therefore, we use an ND-style representation, where each inference consists of a set of premises $P_1,\dots,P_n$ and a set of conclusions $C_1,\dots,C_m$. Each premise has attached a possibly empty set of hypothesis $\mathcal{H}_i$, as depicted below:

$$\frac{\begin{array}{ccc}[\mathcal{H}_1] & & [\mathcal{H}_m]\\ \vdots & & \vdots \\ P_1 & \dots & P_m\end{array}}{C_1 \quad \dots \quad C_n} \tag{6.12}$$

Intuitively, such an inference corresponds to the assertion

$$[(\mathcal{H}_1 \Rightarrow P_1) \wedge \dots \wedge (\mathcal{H}_m \Rightarrow P_m)] \Rightarrow (C_1 \wedge \dots \wedge C_n) \tag{6.13}$$

In general, any assertion can be transformed to an inference of the form (6.12). The form of these inferences has the advantage that it gives an assertion a richer structure, as it divides the formula into premises and conclusions. Based on this structure, further control information and search algorithms can conveniently be defined (see Chapter 9 for details).

Before presenting a general algorithm that computes the inference form for a given assertion, we illustrate the correspondence between inferences and assertions by several examples. Similar to sequents, we will consider special window structures on the free variable tree corresponding to the assertion. Let us note that assertions denote known facts and therefore have negative polarity.

Example 6.2.1. *The following examples show an assertion together with polarities and uniform types on the left, as well as the corresponding inference representation on the right. To be able to differentiate between γ and δ-variables, we denote the former by upper-case letters and the latter by lower case letters. Subformulas corresponding to premises and conclusions are identified by windows w_i.*

$$\left[\underbrace{(x \in A \cap B)^+}_{w_1^+} \Rightarrow^\beta \underbrace{\left((x \in A)^- \wedge^\alpha (x \in B)^-\right)^-}_{w_2^-}\right]^- \qquad \frac{x \in A \cap B}{x \in A \quad x \in B} \tag{6.14}$$

$$\left[\underbrace{\left((x \in A)^- \Rightarrow^\alpha (x \in B)^+\right)^+}_{w_1^+} \Rightarrow^\beta \underbrace{(A \subset B)^-}_{w_2^-}\right] \qquad \frac{\begin{array}{c}[x \in A]\\ \vdots \\ x \in B\end{array}}{A \subset B} \tag{6.15}$$

$$\left[\left(\underbrace{\left(A^- \Rightarrow^\alpha C^+\right)^+}_{w_1^+} \wedge^\beta \underbrace{\left(B^- \Rightarrow^\alpha C^+\right)^+}_{w_2^+}\right)^+ \Rightarrow^\beta \left(\underbrace{\left(A^+ \vee^\alpha B^+\right)^+}_{w_3^+} \Rightarrow^\beta \underbrace{C^-}_{w_4^-}\right)^-\right]^- \qquad \frac{\begin{array}{c}[A]\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array} \quad A \vee B}{C} \tag{6.16}$$

$$\left[(\underbrace{A^+}_{w_1^+} \wedge^\beta \underbrace{(B^+ \vee^\alpha C^+))^+}_{w_2^+} \Rightarrow^\beta \underbrace{(A^- \wedge^\alpha B^- \vee^\beta A^- \wedge^\alpha C^-)^-}_{w_3^-}\right]^- \qquad \frac{A \quad (B \vee C)}{A \wedge B \vee A \wedge C} \tag{6.17}$$

The inference representation of an inference can stepwise be computed by an algorithm, which is shown in Figure 6.1. In each step, it refines the set of premises, the hypotheses corresponding to each premise, or the set of conclusions of the assertion that is considered. Each premise, conclusion, or hypothesis corresponds to a particular node of the free variable indexed formula tree of the assertion and is marked by a window.

Initially, an assertion A is put into a window using the INIT rule (more precisely the free variable indexed formula tree corresponding to the assertion). The window is annotated with a subscript C indicating that it corresponds to a conclusion of the inference that is obtained when the algorithm terminates. Windows that are annotated with P correspond to premises of the inference, while windows annotated with H correspond to hypotheses of premises. The INIT rule is only allowed once during the initialization of the algorithm. PREM-I introduces a new premise to a conclusion C if it is of the form $A \Rightarrow B$ and only a single conclusion has been derived so far. The latter restriction is due to the fact that we want all premises to be β-related to all conclusions and the conclusions α-related to each other. SPLIT-C splits a conclusion of the form $A \wedge B$, thereby introducing multiple conclusions. For backward application this has the advantage that the subformulas can be independently matched, which corresponds to $\wedge$ commutativity. SPLIT-P splits premises as long as their corresponding set of hypotheses is empty. SPLIT-H decomposes a conjunctive hypothesis into two hypothesis. Finally, HYP-I introduces a new hypothesis to a premise, if the premise is of the form $A \Rightarrow B$.

For convenience, we introduce the following terminology:

Notation 6.2.2. *Let (S, f) be an annotated substructure and let S' be a substructure of S with label $\varphi(A, B)$, such that there exists an $n \in dom(f)$ with $f(n) = S'$. We use the following notation to express the replacement of the window n by two windows n' and n'' that point to the direct substructures with label A and B of S:*

$$\frac{\boxed{A} \quad \boxed{B}}{\boxed{\varphi(A, B)}} \tag{6.18}$$

The replacement is further controlled by annotations on the windows. A window w containing the formula φ which is annotated with an annotation A is denoted by $\boxed{\varphi}_A$.

Definition 6.2.3 (Premise Node, Conclusion Node)**.** *Let A be an assertion and let A' be the result of applying the rules of Figure 6.1 to A^-.*

- *The nodes of A that are annotated with $\boxed{n}_P$ in A' are called proper premise nodes of A.*
- *The nodes of A that are annotated with $\boxed{n}_H$ in A' are called hypotheses of A. The hypotheses corresponding to a proper premise p_i are denoted by $\mathsf{Hyps}(p_i)$.*
- *The maximal node with respect to $\prec$ that contains p_i and all its hypotheses is denoted by* $\mathrm{node}_P(P_i)$ *and called* complete premise node*.*
- *The nodes of A that are annotated $\boxed{n}_C$ are called conclusions of A.*

$$\frac{\boxed{A^-}_C}{A^-}\ \textsc{init} \qquad \boxed{\cdot}_C = 1\ \frac{\boxed{A^+}_P \quad \boxed{B^-}_C}{\boxed{(A \Rightarrow B)^-}_C}\ \textsc{prem-i} \qquad \frac{\boxed{A^-}_C \quad \boxed{B^-}_C}{\boxed{(A \wedge B)^-}_C}\ \textsc{split-c}$$

$$\mathsf{Hyps}(A \wedge B) = \emptyset\ \frac{\boxed{A^+}_P \quad \boxed{B^+}_P}{\boxed{(A \wedge B)^+}_P}\ \textsc{split-p} \qquad \frac{\boxed{A^-}_H \quad \boxed{B^-}_H}{\boxed{(A \wedge B)^-}_H}\ \textsc{split-h}$$

$$\frac{\boxed{A^-}_H \Rightarrow \boxed{B^+}_P}{\boxed{(A \Rightarrow B)^+}_P}\ \textsc{hyp-i}$$

Figure 6.1: Inference Rules

Remark 6.2.4. *A conclusion $(A \vee B)^-$ is not processed any further. One possibility would be to generate two inferences, one with conclusion A and additional premise $\neg B$ and vice versa. Similarly, we do not split a hypothesis $(A \vee B)^-$. Note that it would also be possible to introduce a case split automatically (possibly transforming the assertion using the Schütte rule, see Section 5.2.7 for details).*

Theorem 6.2.5. *For any assertion A, the algorithm specified in Figure 6.1 has the following properties:*

(i) it terminates with a unique result.

(ii) the resulting window structure is spanning for A .

(iii) all conclusion windows have negative polarity and are α-related to each other .

(iv) all premise windows have positive polarity and are β-related to each other and β-related to all conclusions.

(v) all hypotheses have negative polarity and are α-related to their corresponding premise.

(vi) Let c be the node that is maximal with respect to $\prec$ and contains all conclusions. Let $R_1, \ldots, R_n$ be all maximal nodes that are below the root node and β-related to c. Moreover, let $P_1, \ldots, P_n$ denote the proper premises of the assertion A. Then

$$\overline{\beta}(R_1, \ldots, R_n) = \textbf{\textit{Label}}(\text{node}_P(P_1)) \wedge \ldots \wedge \textbf{\textit{Label}}(\text{node}_P(P_n)) \tag{6.19}$$

Proof. First, we prove (i). Let n be a node of the free variable indexed formula tree. We define the weight of n as

$$|n| := \begin{cases} 1 + \sum_{n' \in childs(n)} |n'| & \text{if } childs(n) \neq \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{6.20}$$

Moreover, let f be a window structure. The weight of f is defined as

$$|f| := \sum_{l \in dom(f)} |f(l)| \tag{6.21}$$

As $|f|$ is bounded below and each application of a rule – except the INIT rule which is only applied once – decreases $|f|$, the algorithm terminates. Moreover, as there are no critical pairs, the result is unique.

Let us now consider the properties (ii)-(vi). Obviously, all these properties hold initially, as A has negative polarity and there is only one $n \in dom(f)$ that points to the root node of the assertion A, i.e., $f(n) = Q$ with $\textbf{Label}(Q) = A$. It is sufficient to show that each rule application preserves these properties. The result then follows by induction over the length of the derivation. We show exemplarily the proof of property (vi):

Property (vi): Let A' denote the situation after n rule applications to $\boxed{A}$, i.e., $\boxed{A} \rightarrow^n A'$. By induction hypothesis, we can assume that (6.19) holds. We perform a case distinction on the next rule which is applied:

PREM-I**:** Let C denote the unique conclusion to which PREM-I is applied to and which has the form $A \Rightarrow B$. By induction hypothesis, we know that (6.19) holds for C. Applying the rule PREM-I to $A \Rightarrow B$ results in the new conclusion B. The maximal nodes that are below the root node and β-related to B are those of C with the additional node A which has become a new premise. Therefore, (6.19) holds in $\boxed{A} \rightarrow^{n+1} A''$.

SPLIT-C**:** No premise or hypothesis is changed. Moreover, the maximal node with respect to $\prec$ that contains all conclusions remains the same. Therefore, (6.19) holds in $\boxed{A} \rightarrow^{n+1} A''$, provided that it held in $\boxed{A} \rightarrow^{n} A'$.

SPLIT-P**:** Let $P_1, \ldots, P_n$ denote the set of premises before the application of the rule, i.e., in the state $\boxed{A} \rightarrow^n A'$. By induction hypothesis, (6.19) holds in A'. Let $A \overset{\text{SPLIT-P}}{\longrightarrow} A''$. Without loss of generality, we can assume that SPLIT-P is applied to P_n. Then, the premises in A'' are $P_1, \ldots, P_{n-1}, Q_1, Q_2$ with $Q_1 \wedge Q_2 = P_n$.

SPLIT-H**,**HYP-I**:** The splitting or introduction of hypotheses does not change the set of premises.

□

Example 6.2.6. *The following example shows the derivation of the inference rule* (6.16) *as a sequence of the window transformation steps shown in Figure 6.1.*

$$\boxed{\left[\left((A^- \Rightarrow^\alpha C^+)^+ \wedge^\beta (B^- \Rightarrow^\alpha C^+)^+\right)^+ \Rightarrow^\beta \left((A^+ \vee^\alpha B^+)^+ \Rightarrow^\beta C^-\right)^-\right]^-}_C \tag{6.22}$$

$$\to \left[\boxed{\left((A^- \Rightarrow^\alpha C^+)^+ \wedge^\beta (B^- \Rightarrow^\alpha C^+)^+\right)^+}_P \Rightarrow^\beta \boxed{\left((A^+ \vee^\alpha B^+)^+ \Rightarrow^\beta C^-\right)^-}_C\right]^- \tag{6.23}$$

$$\to \left[\left(\boxed{(A^- \Rightarrow^\alpha C^+)^+}_P \wedge^\beta \boxed{(B^- \Rightarrow^\alpha C^+)^+}_P\right)^+ \Rightarrow^\beta \left(\boxed{(A^+ \vee^\alpha B^+)^+}_P \Rightarrow^\beta \boxed{C^-}_C\right)^-\right]^- \tag{6.24}$$

$$\to \left[\left(\left(\boxed{A^-}_H \Rightarrow^\alpha \boxed{C^+}_P\right)^+ \wedge^\beta \left(\boxed{B^-}_H \Rightarrow^\alpha \boxed{C^+}_P\right)^+\right)^+ \Rightarrow^\beta \left(\boxed{(A^+ \vee^\alpha B^+)^+}_P \Rightarrow^\beta \boxed{C^-}_C\right)^-\right]^- \tag{6.25}$$

This gives rise to the following definition:

Definition 6.2.7 (Inference Representation of Assertions)**.** *Let A be an assertion with premise nodes $\{p_1, \ldots, p_n\} =: \mathcal{P}$ and conclusion nodes $\{c_1, \ldots, c_m\} =: \mathcal{C}$. Let $\mathcal{L} = \{l_1, \ldots, l_{n+m}\}$ be disjoint labels. Then $I : \mathcal{L} \to \langle \mathcal{P} \cup \mathcal{C} \rangle$ is an* inference.

We write

$$\frac{\begin{array}{ccc}[\mathsf{Hyps}(p_1)] & & [\mathsf{Hyps}(p_n)] \\ \vdots & & \vdots \\ l_1 : p_1 & \ldots & l_n : p_n\end{array}}{l_{n+1} : c_1 \ldots l_{n+m} : c_m} \tag{6.26}$$

$p_1, \ldots, p_n$ are called premises of the inference, $c_1, \ldots, c_n$ conclusions of the inference, and $\mathsf{Hyps}(p_i)$ the hypothesis corresponding to the premise p_i.

Notation 6.2.8. *Given an inference $\mathcal{I}$ and a premise label p we write $\mathcal{I}(\overline{p})$ to denote the complete premise node corresponding to p, i.e., the node that includes the proper premise and all hypotheses. The notation is borrowed from topology where $\overline{S}$ denotes the closure of S.*

6.2.1 Preprocessing

To get more natural inferences, two preprocessing steps are necessary. Consider a proper definition, such as the definition of subset:

$$\forall A, B.A \subset B \Leftrightarrow \forall x.x \in A \Rightarrow x \in B \tag{6.27}$$

which cannot be further processed, as we cannot assign polarities underneath the equivalence. This can be solved either by transforming the assertion before inserting it into the indexed formula tree, or by applying the Leibniz equality to obtain both directions of the equivalence

$$\forall A, B.A \subset B \Rightarrow \forall x.x \in A \Rightarrow x \in B \tag{6.28}$$

$$\forall A, B.\forall x.x \in A \Rightarrow x \in B \Rightarrow A \subset B \tag{6.29}$$

For equivalences, we also keep the original formula to obtain a rewrite rule, as shown in Section 6.6.

The second optimization deals with assertions that have been fused together using a conjunction, such as

$$[\forall x.even(x) \vee odd(x)] \wedge [\forall x.odd(x) \Rightarrow \neg even(x)] \tag{6.30}$$

which corresponds to the two assertions

$$\forall x.even(x) \vee odd(x) \tag{6.31}$$
$$\forall x.odd(x) \Rightarrow \neg even(x) \tag{6.32}$$

Those can be detected by the property that the subformulas do not share any context. Again, they can be handled either within the framework, or within a preprocessing step. Note however that such assertions usually do not occur in practice.

Another preprocessing, that can be performed independently of the actual proof search, is the preprocessing of quantifiers. Baaz and Leitsch [BL94] have shown that prenexing a formula, i.e., pulling the quantifiers out, makes its proof longer, as introduced Skolem functions might get additional arguments. Therefore, quantifiers should be pulled in, which can be achieved by a simplification phase before the actual proof starts, using the following set of rewrite systems:

$$(\forall x.P(x) \wedge Q) \equiv (\forall x.P(x)) \wedge Q \qquad \text{if } x \notin \mathcal{FV}(Q) \tag{6.33}$$
$$(\forall x.P(x) \vee Q) \equiv (\forall x.P(x)) \vee Q \qquad \text{if } x \notin \mathcal{FV}(Q) \tag{6.34}$$
$$(\exists x.P(x) \wedge Q) \equiv \exists x.P(x) \wedge Q \qquad \text{if } x \notin \mathcal{FV}(Q) \tag{6.35}$$
$$(\exists x.P(x) \vee Q) \equiv \exists x.P(x) \vee Q \qquad \text{if } x \notin \mathcal{FV}(Q) \tag{6.36}$$

In the sequel, we will assume that assertions are always preprocessed.

6.3 Assertion Application

As we have seen in the previous section, inferences are just another representation of assertions. In this form, each assertion consists of multiple premises and conclusions, augmented by a possibly empty set of hypotheses for each premise (see Definition (6.2.7)). Each of these premises, conclusions, and hypotheses can be identified with a specific node in the free variable task tree.

When applying an inference to transform a given task, we typically try to instantiate some of its formal arguments, i.e., try to find corresponding subformulas in a task and a substitution σ which makes the formula in the task and the formula corresponding to the formal argument of the inference equal. Thereby additional conditions which result from the deep access to subformulas have to be respected, such as restricting matching candidates for premises to subformulas with negative polarity.

From an abstract point of view to determine how a single inference can be applied can be understood as a classical retrieval problems of *candidate terms* from a given *query term* or *subject term* satisfying a specific relationship. Such problems occur frequently in the setting of automated theorem proving, and powerful indexing techniques have been developed to support efficient retrieval. However, in contrast to the standard approach, we have to solve a simultaneous unification/matching problem, as several premises/conclusions are usually instantiated. Such a problem is known in the context of hyperresolution, where one also has to solve the problem of simultaneous retrieval of unifiable terms. Instead of matching/unifying all inference nodes simultaneously, a reasonable strategy consists of matching one inference node at a time, starting from an inference where no node has been matched. This has been done for example in Vampire [RV99].

We follow a similar approach and introduce the notion of *partial argument instantiations* (PAI), which is an adaptation of the notion from [BS01b] to the new inferences presented here and makes the instantiation process explicit. The idea is as follows: We start with a PAI which maps all elements of the inference to $\perp$. Subsequently, this partial argument instantiation is updated by instantiating a previously uninstantiated argument of the inference, resulting in a new PAI. We say that the new PAI is an *partial argument instantiation update* with respect to the previous PAI.

Let us now start with the development of the formal framework. First, we introduce the notion of a *task position*, which represents the matching candidates of a specific proof task, i.e., nodes of the task tree that are labeled with formulas. The corresponding positions are split into three sets, according to the polarities that can arise.

Definition 6.3.1 (Task Position)**.** *Let $T = w_1, \ldots, w_n$ be a task. The task positions of T, denoted by $\mathcal{P}os$, are defined to be the following set:*

$$\mathcal{P}os(T) = \bigcup_{i=1}^{n} \mathcal{S}(w_i) \tag{6.37}$$

Moreover, we define the following three subsets:

$$\begin{aligned}\mathcal{P}os(T)^+ &= \{t \in \mathcal{P}os(T) \mid pol(t) = +\} && (6.38)\\ \mathcal{P}os(T)^- &= \{t \in \mathcal{P}os(T) \mid pol(t) = -\} && (6.39)\\ \mathcal{P}os(T)^\circ &= \{t \in \mathcal{P}os(T) \mid pol(t) = \circ\} && (6.40)\\ & && (6.41)\end{aligned}$$

Example 6.3.2. *The task*

$$A^-, \left(B^- \wedge^\alpha C^-\right)^- \vdash^\alpha D^+ \tag{6.42}$$

has the following task positions:

- $\mathcal{P}os(T) = \{A, B, C, B \wedge C, D\}$
- $\mathcal{P}os(T)^- = \{A, B, C, B \wedge C\}$
- $\mathcal{P}os(T)^+ = \{D\}$
- $\mathcal{P}os(T)^0 = \emptyset$

Remark 6.3.3. *Note that it is not allowed to match above windows, even though the labels of the corresponding nodes principally denote admissible matching positions. In particular, nodes with labels A, B which do occur in two different windows cannot be matched against $A \wedge B$, even if their parent node has label $A \wedge B$.*

Based on the notion of a task position, we now define the notion of an *inference substitution* with respect to an inference $\mathcal{I}$ and a task T. An inference substitution consists of two components: a partial mapping $\sigma_{\mathcal{N}} : \mathcal{L} \hookrightarrow \mathcal{P}os(T)$ that identifies a task position for a formal argument $l \in \mathcal{L}$, and a substitution σ. Intuitively, an instantiated argument $l \in \mathcal{L}$ identifies two nodes in the free variable task tree – one corresponding to the node of the inference, and one corresponding to a node of the task – that are made equal under σ:

$$\sigma_{\mathcal{N}}(l)\sigma = \mathcal{I}(l)\sigma \tag{6.43}$$

Definition 6.3.4 (Inference Substitution)**.** *Let I be an inference with names $\mathcal{L}$ for premises and conclusions and let T be a task. An* inference substitution wrt. *T is a pair $\sigma_{\mathcal{I}} = \langle \sigma_{\mathcal{N}}, \sigma \rangle$ consisting of a mapping $\sigma_{\mathcal{N}} : \mathcal{L} \hookrightarrow \mathcal{P}os(T)$ and a substitution σ. Given an inference substitution, its domain is defined as*

$$dom(\sigma_{\mathcal{I}}) := \{l \in \mathcal{L} \mid \sigma_{\mathcal{N}}(l) \neq \bot\} \tag{6.44}$$

Given an inference and a corresponding inference substitution, its application will be modeled by a sequence of resolution replacement rules within the Core calculus (the exact details are explained in the next section). Thereby, each underlying resolution replacement rule application is subject to several conditions on the underlying free variable task tree. The essence of the following definition, which looks rather technical at a first glance, is to lift these conditions to the level of inferences, thereby taking the structural properties of task trees into account (c.f. Theorem 6.1.12). That is, once an inference substitution satisfies the specified conditions, all induced resolution replacement rules are admissible.

Definition 6.3.5 (Partial Argument Instantiation)**.** *Let $\sigma_{\mathcal{I}}$ be an inference substitution with respect to an inference $\mathcal{I}$ some task T. We say that $\sigma_{\mathcal{I}}$ is a* partial argument instantiation *iff*

(i) σ is an admissible substitution.

(ii) for all $l \in dom(\sigma_{\mathcal{I}})$ it holds

- *if l denotes a premise, then $\sigma_{\mathcal{I}}(l) \in \mathcal{P}os^{-}(T)$ and $\textbf{Label}(\sigma_{\mathcal{I}}(l))\sigma = \textbf{Label}(l)\sigma$. Moreover, for any conclusion c $\sigma_{\mathcal{I}}(c)$ and $\sigma_{\mathcal{I}}(l)$ are α-related.*
- *if l denotes a conclusion, then $\sigma_{\mathcal{I}}(l) \in \mathcal{P}os^{+}(T)$ and $\textbf{Label}(\sigma_{\mathcal{I}}(l))\sigma = \textbf{Label}(l)\sigma$.*
- *for two conclusion labels $l, l' \in dom(\sigma_{\mathcal{I}})$, $\sigma_{\mathcal{I}}(l)$ and $\sigma_{\mathcal{I}}(l')$ are strictly β-related to each other.*

Moreover, we require

(iii) no position is contained within another position, i.e., for all $l, l' \in dom(\sigma_{\mathcal{I}}^{T})$ it holds that $l \not\prec l'$.

(iv) for two premise labels $l, l' \in dom(\sigma_{\mathcal{I}}^{T})$, it holds that $\sigma_{\mathcal{I}}^{T}(l)$ and $\sigma_{\mathcal{I}}^{T}(l')$ are α-related to each other.

A PAI $\sigma_{\mathcal{I}}$ is called empty, *iff $dom(\sigma_{\mathcal{I}}) = \emptyset$.*

Let us explain the additional conditions (iii) and (iv), which are not needed to ensure the correctness of the inference application, before giving an example of a PAI. The condition (iv) corresponds to the intuition of not using assertions from different branches simultaneously, as the following example illustrates:

Example 6.3.6. *Consider the inference* conj-I

$$\textsc{conj-I}\ \frac{p_1 : A \quad p_2 : B}{c_1 : A \wedge B} \tag{6.45}$$

and the task T

$$\left[A^{-} \vee^{\beta} B^{-}\right]^{-} \vdash \left[A^{+} \wedge^{\beta} B^{+}\right]^{+} \tag{6.46}$$

Suppose that the conclusion has been matched against the formula $A \wedge B$. Without the condition, we could subsequently both match p_1 and p_2 in (6.46)*, thereby introducing the proof obligations $(\neg B)^+$ and $(\neg A)^+$. Finally, condition* (iii) *ensures that all contraction steps need to be performed explicitly. Consider again the inference* CONJ-I *shown above and the task*

$$A^- \vdash B^+ \tag{6.47}$$

Without the condition (iii)*, we can apply* CONJ-I *in forwards direction to deduce $A \wedge A$, which is not possible with the restriction.*

Let us now illustrate the concept of a PAI:

Example 6.3.7. *Consider the task*

$$\left[(A \subset B)^+ \Rightarrow^\beta (f(A) \subset f(B))^-\right]^- \vdash \mathit{Auto}(f,G)^+ \Rightarrow^\alpha f(\mathit{Ker}(f,G)) \subset G^+ \tag{6.48}$$

and the inference

$$\frac{p_1 : U \subset V \quad p_2 : V \subset W}{c : U \subset W}\,\textsc{Trans}\subset \tag{6.49}$$

For premises, the nodes corresponding to the following formulas are candidates for the instantiation process:

$$\{f(A) \subset f(B), \mathit{Auto}(f,G)\} \tag{6.50}$$

Similarly, for the conclusion of the inference the following nodes are candidates:

$$\{f(\mathit{Ker}(f,G)) \subset G, A \subset B\} \tag{6.51}$$

Using the substitution

$$\sigma = \{U \mapsto f(\mathit{ker}(f,G)), W \mapsto G, A \mapsto \mathit{ker}(f,G), V \mapsto f(G)\} \tag{6.52}$$

the task reads as

$$\mathit{ker}(f,G) \subset B \Rightarrow (f(\mathit{ker}(f,G)) \subset f(B)) \vdash \mathit{Auto}(f,G) \Rightarrow f(\mathit{Ker}(f,G)) \subset G \tag{6.53}$$

and the inference as

$$\frac{f(\mathit{ker}(f,G)) \subset f(G) \quad f(G) \subset G}{f(\mathit{ker}(f,G)) \subset G} \tag{6.54}$$

Therefore, we can both instantiate the first premise and the conclusion of the inference, obtaining the following PAI which consists of σ and the inference substitution

$$\sigma_{\mathcal{L}} : \{p_1, p_2, c\} \to \mathcal{P}os^{(T)};\ \sigma_{\mathcal{L}}(x) = \begin{cases} f(\mathit{ker}(f,G)) \subset f(B) & x = p_1 \\ \bot & x = p_2 \\ \mathit{Ker}(f,G) \subset B & x = c \end{cases} \tag{6.55}$$

The matching process is illustrated in Figure 6.2, which shows the free variable task tree. In the figure, solid boxes indicate the corresponding positions within the task tree. The dashed boxes correspond to proof obligations that will be introduced when applying the inference. The inference substitution (6.55) *is admissible, because*

(i) σ is admissible

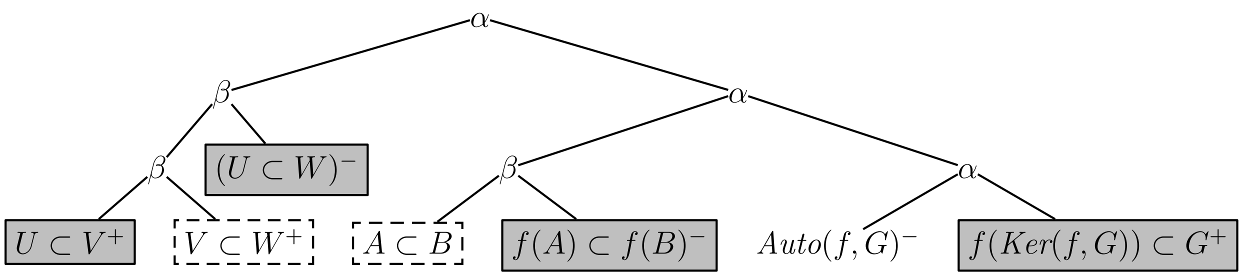


Figure 6.2: Initial free variable indexed formula tree for our example

(ii)
- *$\sigma_{\mathcal{I}}(p_1)$ has negative polarity and is α-related to $\sigma_{\mathcal{I}}(c)$ via the right child of the root node; moreover, $\sigma_{\mathcal{I}}(p_1)\sigma = \mathcal{I}(p_1)\sigma$*
- *$\sigma_{\mathcal{I}}(c)$ has positive polarity; moreover, $\sigma_{\mathcal{I}}(c)\sigma = \mathcal{I}(c)\sigma$*
- *there is only one conclusion*

(iii) the task positions do not overlap

(iv) there is only one instantiated premise, so all instantiated premises are α-related to each other

Notation 6.3.8. *From now on, we will denote partial argument instantiations with respect to an inference $\mathcal{I}$ and a task T by $pai^T_{\mathcal{I}}$.*

Motivated by the above example, we now introduce the notion of a partial argument instantiation update to make the stepwise search process of a PAI explicit. The intuition of a PAI update for a given PAI is that at least one new task position has been added to the PAI. Formally, this is captured by the following definition:

Definition 6.3.9 (Partial Argument Instantiation Update)**.** *Let $\mathcal{I}$ be an inference, T be a task, $pai^T_{\mathcal{I}}$, $pai'^T_{\mathcal{I}}$ be partial argument instantiations for $\mathcal{I}$ with respect to T. Then $pai'^T_{\mathcal{I}}$* is a partial argument update of *$pai^T_{\mathcal{I}}$ iff*

- *$pai^T_{\mathcal{I}}(l) = pai^T_{\mathcal{I}}(l)$ for all $l \in dom(pai^T_{\mathcal{I}})$.*
- *there is at least one* formal argument *of $\mathcal{I}$ in $dom(pai'^T_{\mathcal{I}}) \setminus dom(pai^T_{\mathcal{I}})$.*

Example 6.3.10. *Let us illustrate the above definition using the inference and task from Example 6.3.7 by showing one possible sequence of PAI updates leading from the empty PAI to the one shown in* (6.55)*.*

$$\longrightarrow\ \sigma_{\mathcal{L}}(x) = \begin{cases} \bot & x = p_1 \\ \bot & x = p_2 \\ Ker(f,G) \subset B & x = c \end{cases} \longrightarrow\ \sigma_{\mathcal{L}}(x) = \begin{cases} f(ker(f,G)) \subset f(B) & x = p_1 \\ \bot & x = p_2 \\ Ker(f,G) \subset B & x = c \end{cases}$$

The instantiation process can also be shown schematically by abstracting over the concrete statements, resulting in a so-called PAI-status*. This allows the computation of all possible status updates statically. For the inference* (6.49)*, the corresponding graph is shown in Figure 6.3, where the rounded boxes correspond to PAIs and indicate the instantiated arguments. The squared boxes indicate the argument that is added when following the edge.*

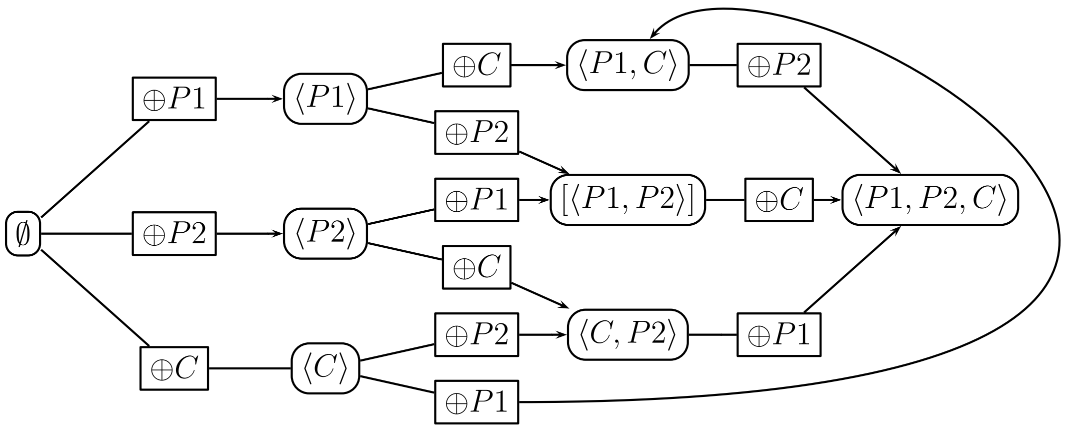


Figure 6.3: Possible status updates for the inference (6.49)

To be able to describe the effects of the application of an inference, let us introduce three main directions in which an inference can be applied: *forwards*, *backwards*, and *close*. Intuitively, an inference is applied in forwards direction if it introduces new facts, in backwards direction if it reduces a goal to new subgoals, and in close direction if it closes the goal.

Definition 6.3.11 (Applicable Inference, Forward/Backward/Close Direction)**.** *Let $\mathcal{I}$ be an inference, T be a task and $pai_{\mathcal{I}}^{T}$ a partial argument instantiation. $pai_{\mathcal{I}}^{T}$ is applicable iff one of the following conditions hold:*

(i) for all conclusions c, $pai_{\mathcal{I}}^{T}(c) \neq \perp$

(ii) there is a premise p such that $pai_{\mathcal{I}}^{T}(p) \neq \perp$.

We classify $pai_{\mathcal{I}}^{T}$ to the following directions: $pai_{\mathcal{I}}^{T}$ is called

- forward partial argument instantiation *if for all $c \in C$ it holds that $pai_{\mathcal{I}}^{T}(c) = \perp$ and there is some $p \in P$ such that $pai_{\mathcal{I}}^{T}(p) \neq \perp$.*
- backward partial argument instantiation *if there exists a $c \in C$ such that $pai_{\mathcal{I}}^{T}(c) \neq \perp$.*
- closing partial argument instantiation *if all premises and all conclusions are instantiated.*

In the following, we describe the application of such an inference to a given proof state and show how a given task is transformed. We will define two transformations, dependent on whether the inference is applied forwards or backwards.

6.4 Assertions: Backward Application

When applying an inference in the backward direction, all conclusions of the inference need to be instantiated. These conclusions are then replaced by some new subgoals resulting from the application of the inference. Internally, the application of an inference is reduced to a sequence of resolution replacement rules in the CORE calculus. As this application intuitively consumes some parts of the free task tree, the multiplicities of the involved quantifiers of the inference are additionally increased, such that the inference can be applied several times. Finally, the consumed part is removed from the indexed formula tree using weakening. To get an intuitive understanding of the underlying operations, let us illustrate the induced resolution replacement rules by means of an example.

Example 6.4.1. *We have already seen in Example 6.3.7 how the inference* TRANS- $\subset$ *can be instantiated by unifying the conclusion with the subformula $f(Ker(f,G)) \subset G$ and the first premise with the subformula $f(A) \subset f(B)$. Applying the inference reduces the task*

$$A \subset B \Rightarrow f(A) \subset f(B) \vdash Auto(f,G) \Rightarrow f(ker(f,G)) \subset G \tag{6.56}$$

in one step to

$$A \subset B \Rightarrow (f(A) \subset f(B)) \vdash Auto(f,G) \Rightarrow (Ker(f,G) \subset B \land f(B) \subset G) \tag{6.57}$$

which can be proved immediately using the definitions of Auto and Ker, instantiating B with G.

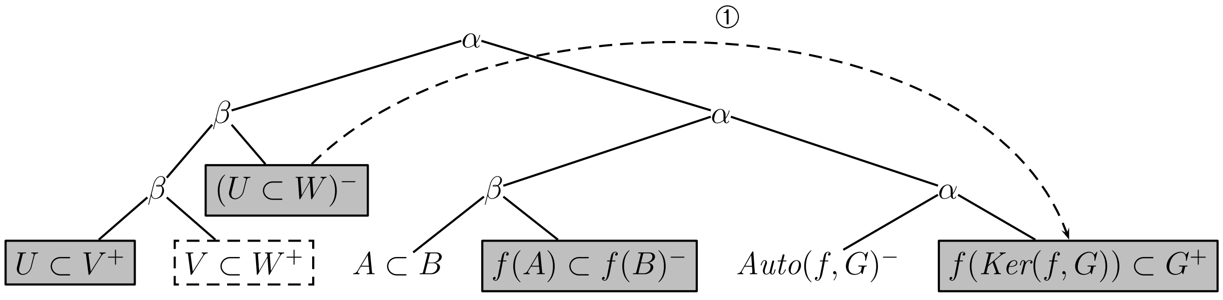
The modified task (6.57) is obtained by the application of two resolution replacement rules, each of which is induced by an instantiated premise/conclusion. More precisely, we consider the two node positions of the free variable task tree corresponding to the inference node and the task node of the instantiated formal argument. For the conclusion, this pair consists of the inference node $(U \subset W)$ and the task node $f(ker(f,G)) \subset G$. The requirements of Definition 6.3.5 guarantee that this indeed induces a resolution replacement rule, which allows the replacement of the task node with the conditions of the conclusion node of the inference:

$$U \subset W^- \to \langle (U \subset V \land V \subset W)^+ \rangle \tag{6.58}$$

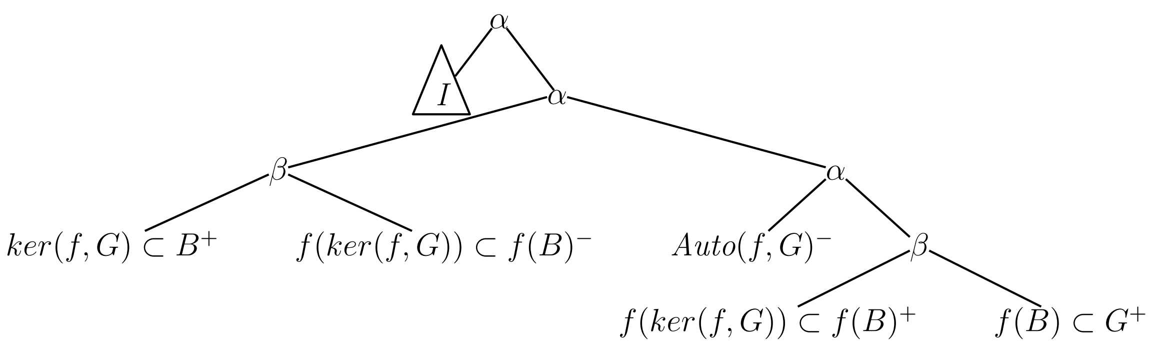
which is composed of the two premises and equals $\beta(U \subset V^+, V \subset W^+)$ (c.f. Theorem 6.2.5). Under the instantiation, the rule becomes

$$(f(Ker(f,G)) \subset G)^+ \to \langle (f(Ker(f,G)) \subset V \land V \subset G)^+ \rangle \tag{6.59}$$

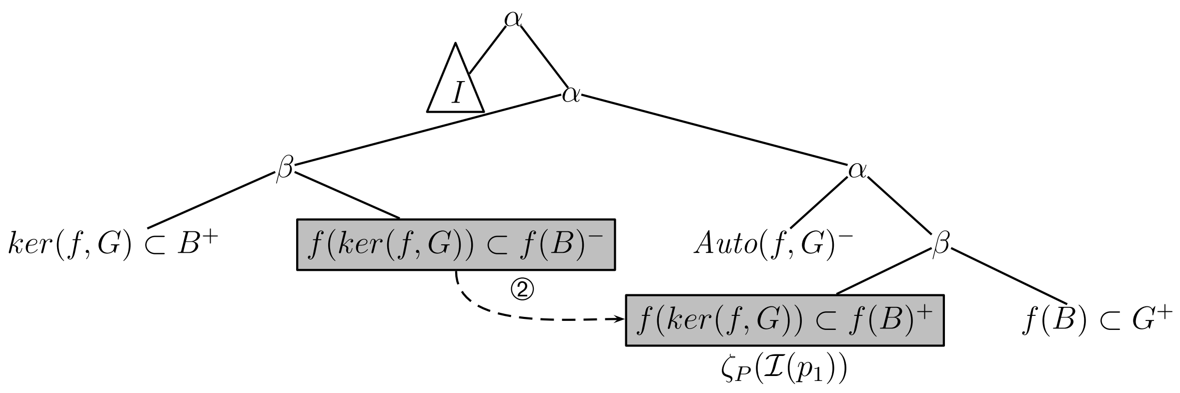
Similarly, the instantiated premise induces a resolution replacement rule. However, for backward inferences, the resolution replacement rule belonging to the conclusion of the inference is applied first, as indicated in the subsequent picture by ①:



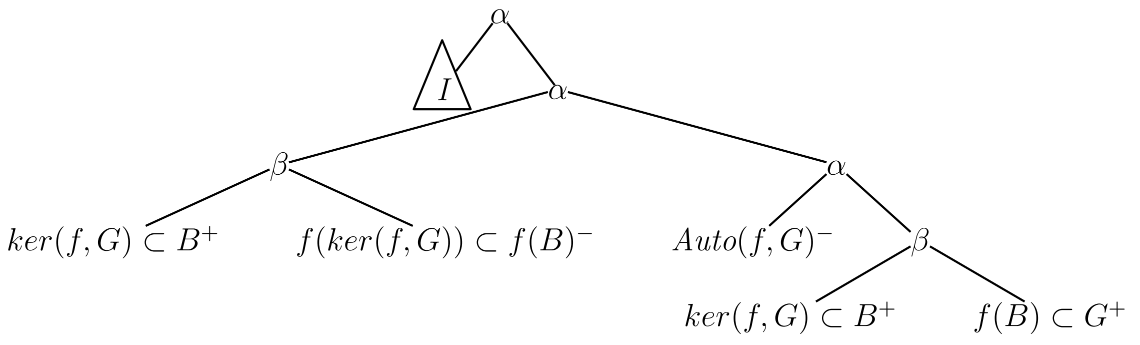
To enable the application of the resolution replacement rule, the involved labels must be equal. Therefore, we have to apply the substitution σ before applying the resolution replacement rule. Afterwards, the replacement can be carried out and results in the following free variable task tree:



As shown above, all premise nodes of the inference have been transferred to the task part of the free variable task tree. More precisely, it is possible to define a mapping ζ_P that identifies for each premise of the inference a substructure of the replaced conclusion: In the figure above, $f(ker(f,G)) \subset f(B)$ corresponds to the first premise, and $f(B) \subset G$ corresponds to the second premise. The next step, indicated by ②, consists of the application of the resolution replacement rule induced by the instantiated premise. The replacement takes place between the task position of the premise, i.e., $pai^T_{\mathcal{I}}(l)$, and the transferred premise node, i.e., $\zeta_P(\mathcal{I}(l))$, as indicated below:



This already produces the overall result of the inference application, namely the task (6.57), *which corresponds to the following task tree:*



As some of the free variables of the inference tree have been instantiated, the inference tree is of no further use and is removed by the application of the weakening rule (see Definition 5.2.17 for details).

The process depicted above can be generalized to arbitrary inferences, for which it works as follows:

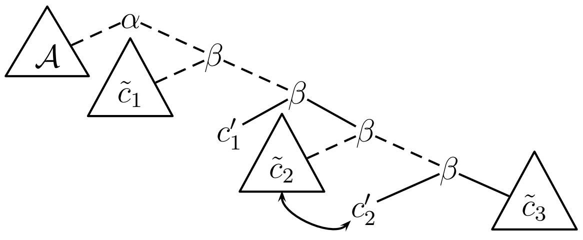
1. Increase the multiplicity of all involved meta-variables of the inference.

2. Apply the resolution replacement rule to the conclusion of the inference, i.e., the replacement rule between the nodes
$$\mathcal{I}(c), pai^T_{\mathcal{I}}(c) \tag{6.60}$$
where $pai^T_{\mathcal{I}}(c)$ is replaced.

3. For each instantiated premise p apply its induced resolution replacement rule, i.e., the replacement rule between the nodes
$$pai^T_{\mathcal{I}}(p), \zeta_P(\mathcal{I}(p)) \tag{6.61}$$
where $\zeta_P(\mathcal{I}(p))$ is replaced.

4. Weaken the formula to clean the task tree.

Notice that the recipe above involves only a single conclusion, while we have defined inferences more generally to also support multiple conclusions. However, as we will show below, it is sufficient to consider a single conclusion, because it is possible to reduce the multi-conclusion case to the single-conclusion case. To that end, we consider an inference with two conclusions c_1 and c_2, and a task with corresponding formulas c'_1 and c'_2 which are possibly embedded in some other β-related nodes $\tilde{c}_1, \ldots, \tilde{c}_n$:



The idea is to use commutativity of $\wedge$ to transform the goal as indicated above and to replace both conclusions simultaneously. On the free variable task tree, the commutativity of $\wedge$ corresponds to a permutation of β-nodes:

Theorem 6.4.2. *Let Q be a free variable indexed formula tree of the form $\beta(C_1, \ldots, C_N)$. Then Q is provable iff the permutation $Q' = \beta(\pi(C_1 \ldots, C_N))$ is provable.*

Proof. Clear, as this kind of permutation does not change the set of paths. □

For an arbitrary number of conclusions, the result follows inductively. Therefore, we subsequently assume only a single conclusion C of inferences in the subsequent presentation. We then proceed in three steps: First, we define the overall effect of the backward application of a $pai^T_{\mathcal{I}}$. We then show how the result is modeled by a sequence of resolution replacement rules. Finally, we show that the adaptations of the windows yield a consistent proof state.

To be able to define the proof state transformation for an arbitrary PAI, it is convenient to describe the proof obligation induced by a single premise, independent of whether it has been instantiated or not. In the first case, conditions may arise due to conditions induced by the replacement rule application. As replacement rules induced by instantiated premises are applied after the replacement rule induced by the conclusion, these conditions are between two task positions of the current task, namely the modified conclusion and the task position of the premise.

Definition 6.4.3 (Premise Conditions). *Let $w_1, \dots, w_n \vdash w_{n+1}, \dots, w_m$ be a task, let π^+ be a task position with positive polarity, and let π^- be a task position with negative polarity that is α-related to π^+ via some node R. Then the conditions of π^- with respect to π^+, denoted by $\mathcal{POB}_P(\pi^-, \pi^+)$, are the formulas corresponding to the labels of nodes between π^- and R that are β-related to π^+ and maximal with respect to $\prec$.*

Example 6.4.4. *The premise conditions of $f(A) \subset f(B)$ with respect to the goal $f(ker(f,G)) \subset G$ in task* (6.48) *are $\langle (A \subset B)^+ \rangle$.*

If a premise is not instantiated, the proof obligation is due to the induced replacement rule of the conclusion and consists of the premise formula modified by the inference substitution σ. Putting both observations together, we are able to characterize the new proof obligation of a single premise with respect to a task and an inference as follows:

Definition 6.4.5 (Backward Inference Conditions). *Let $\mathcal{I}$ be an inference with premise labels $p_1, \dots, p_n$, T be a task, and $pai^T_{\mathcal{I}}$ be a partial argument instantiation with respect to $\mathcal{I}$ and T. Let $\zeta_P : \mathcal{L} \to \mathcal{S}(R)$ be the mapping which identifies for each premise its position in the task tree*[1]*. For each label l denoting a premise, the conditions of l are defined as follows:*

$$\mathcal{POB}_{bw}(l) = \begin{cases} \textbf{\textit{Label}}(\mathcal{I}(\bar{l}))\sigma & \text{if } pai^T_{\mathcal{I}}(l) = \bot \\ \beta(\mathcal{POB}_P(pai^T_{\mathcal{I}}(l), \zeta_P(\mathcal{I}(l)))) & \text{otherwise} \end{cases} \tag{6.62}$$

The overall conditions of the $pai^T_{\mathcal{I}}$ are then given by

$$\bigwedge_{i=1}^{n} \mathcal{POB}_{bw}(p_i) \tag{6.63}$$

Note that the inference conditions for the instantiated premises can be statically determined from the task T. However, in contrast to the CORE proof theory they obey a locality property, as they are determined by only inspecting the window containing the corresponding position, whereas in the case of CORE they are determined by the node c that governs the connectable nodes.

Definition 6.4.6 (Backward Inference Rule Application). *Let $\mathcal{I}$ be an inference, T be a task of the task tree (S, f), and $pai^T_{\mathcal{I}}$ be a backward PAI with respect to $\mathcal{I}$ and T. The effect of applying the PAI consists of replacing the conclusion $pai^T_{\mathcal{I}}(c)$ by* (6.63)*, i.e.,*

$$S|_{pai^T_{\mathcal{I}}(c) \leftarrow (S', f')} \tag{6.64}$$

where S' denotes the substructure for (6.63)*, and f' is defined as follows:*

- *If $pai^T_{\mathcal{I}}(c)$ is a succedent formula, i.e., there is a $n \in dom(f)$ with $f(n) = pai^T_{\mathcal{I}}(c)$*
  - *if $|Prems(\mathcal{I})| = 1$ and $\mathsf{Hyps}(p_1) \neq \emptyset$*

$$f'(n) := \begin{cases} \zeta_P(pai^T_{\mathcal{I}}(p) & \text{for exactly one } n \in dom(f') \\ \zeta_P(pai^T_{\mathcal{I}}(h)) & \text{for each hyp } h \text{ and for exactly one } n \in dom(f') \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6.65}$$

[1]The mapping is induced by the application of the resolution replacement rule induced by the conclusion

– *otherwise*

$$f'(n) = pai^T_{\mathcal{I}}(c) \tag{6.66}$$

for exactly one $n \in dom(f')$.

- *Otherwise* $f'(n)$ *undefined for all* $n \in dom(f')$.

The replacement defined above replaces the goal by new subgoals and adapts the windows of the free variable task tree. During this process, there are several design choices of how to add new windows. We found it convenient to make the goal structure as small as possible. That is, instead of introducing a new goal $A \Rightarrow B$, we introduce a new hypothesis (new window) for A, and a new goal B (another window). Note that this corresponds to some kind of proof normalization in ND by the application of $\Rightarrow_I$ rules. However, this is only possible if the modified subformula is the maximal formula of the window that contains it, i.e., a top-level subgoal in the task representation.

Example 6.4.7. *Consider the inference*

$$\cfrac{\begin{array}{c}[x \in U]\\ \vdots\\ x \in V\end{array}}{U \subset V}\ \textsc{Def}\subset \tag{6.67}$$

and the task

$$\vdash A \cap B \subset B \cap A \tag{6.68}$$

Applying the inference (6.67) *transforms the task into*

$$x \in A \cap B \vdash x \in B \cap A \tag{6.69}$$

as the goal is at top-level. In contrast, the same inference applied to

$$\vdash A \cap B \subset B \cap A \wedge B \cap A \subset A \cap B \tag{6.70}$$

is transformed to

$$\vdash (x \in A \cap B \Rightarrow x \in B \cap A) \wedge B \cap A \subset A \cap B \tag{6.71}$$

We show subsequently that the effect of $pai^T_{\mathcal{I}}$ can be modeled by a sequence of resolution replacement rules. However, before proving the main theorem, we state some intermediate lemmas.

Lemma 6.4.8. *Let* $\mathcal{I}$ *be an inference with premises* $p_1, \ldots, p_n$, T *be a task and* $pai^T_{\mathcal{I}}$ *be a backward PAI. The conclusion replacement rule induced by* $pai^T_{\mathcal{I}}$ *is admissible and results in the modified task*

$$pai^T_{\mathcal{I}}(c) \to \bigwedge_{i=1}^{n} \boldsymbol{Label}(\mathcal{I}(\overline{p_i}))\sigma \tag{6.72}$$

Proof. Recall that $\mathcal{I}(c)$ denotes the node corresponding to the conclusion of the inference, and that $pai^T_{\mathcal{I}}(c)$ denotes the node corresponding to the node of the task (see Definition (6.3.5)). Then $\mathcal{I}(c)$ has negative polarity and $pai^T_{\mathcal{I}}(c)$ has positive polarity. Moreover, both positions are α-related via the root node r of the free variable indexed formula tree. This induces an admissible resolution replacement rule

$$\mathcal{I}(c) \to \langle R_1, \ldots, R_n \rangle \tag{6.73}$$

where $R_1, \dots, R_n$ are the nodes that are below r and β-related to $\mathcal{I}(c)$. Moreover,

$$\textbf{Label}(\mathcal{I}(c))\sigma = \textbf{Label}(pai^T_{\mathcal{I}}(c))\sigma \tag{6.74}$$

therefore $\mathcal{I}(c)$ and $pai^T_{\mathcal{I}}(c)$ are connectable. To determine the concrete form of $R_1, \dots, R_n$, by Lemma 6.1.14 it is sufficient to consider the inference tree corresponding to the assertion representing the inference $\mathcal{I}$ to determine $R_1, \dots, R_n$. By Theorem 6.2.5, all R_i correspond to a single premise or a conjunction of premises and have positive polarity. β-insertion therefore results in replacing the subtree of $pai^T_{\mathcal{I}}(c)$ by the subtree with label

$$\left[\bigwedge_{p \in Prems(\mathcal{I})} \textbf{Label}(\mathcal{I}(\overline{p}))\right] \sigma \wedge true^+ \tag{6.75}$$

which can be simplified by means of the simplification rule (see Definition (5.2.19)) to

$$\bigwedge_{p \in Prems(\mathcal{I})} \textbf{Label}(\mathcal{I}(\overline{p}))\sigma \tag{6.76}$$

□

After applying the resolution replacement rule induced by the conclusion, the replaced conclusion is of the form (6.76). In particular, for each premise of the inference we can identify a corresponding position in the task tree, which we denote by $\zeta_P(\mathcal{I}(p))$. We subsequently apply all induced premise resolution replacement rules, that is, resolution replacement rules between the following positions:

$$pai^T_{\mathcal{I}}(p), \zeta_P(\mathcal{I}(p)) \tag{6.77}$$

This results in (6.64) of Definition 6.4.6.

Lemma 6.4.9. *All induced premise resolution replacement rules are admissible. Application of all induced premise resolution replacement rules transforms* (6.76) *to the conditions of* (6.63).

Proof. We show by induction on the number $|inst(P)|$ of instantiated premises that the application of all induced resolution replacement rules transforms (6.76) to the conjunction of all proof obligations, i.e.,

$$\bigwedge_{p \in Prems(\mathcal{I})} \textbf{Label}(\mathcal{I}(\overline{p}) \rightarrow \bigwedge_{p \in Prems(\mathcal{I})} \mathcal{POB}_{\text{bw}}(p) \tag{6.78}$$

Let us first note that none of the premise positions is destroyed by the application of previous replacement rule, as the task positions are not overlapping due to condition (iii) of Definition (6.3.5).

Base Case: If $|inst(P)| = 0$, then $\mathcal{POB}_{\text{bw}}(p) = \textbf{Label}(\mathcal{I}(p))$ and (6.76) has already the desired form.

Step Case: Let us now assume that $|inst(P)| = n$. Let $P_1, \dots, P_{n-1}$ denote the first $n-1$ instantiated premises. By induction hypothesis,

$$\begin{aligned}\bigwedge_{p \in Prems(\mathcal{I})} \textbf{Label}(\mathcal{I}(p)) &= \bigwedge_{p \in Prems(\mathcal{I}) \backslash p'} \textbf{Label}(\mathcal{I}(p)) \wedge \textbf{Label}(\mathcal{I}(p')) &(6.79)\\ &= \bigwedge_{p \in Prems(\mathcal{I}) \backslash p'} \mathcal{POB}_{\text{bw}}(p) \wedge \textbf{Label}(\mathcal{I}(p')) &(6.80)\end{aligned}$$

p' induces an admissible resolution replacement rule $\zeta_P(\mathcal{I}(p')) \to \langle R_1, \ldots, R_n \rangle$ which can be applied to the node $\zeta_P(\mathcal{I}(p'))$. This is because $\zeta_P(\mathcal{I}(p'))$ and $pai^T_{\mathcal{I}}(p')$ have opposite polarity and their labels are equal under the substitution σ. Moreover, they are α-related: If they are contained in the same window, this is ensured by Definition (6.3.5) and the conditions are the β-related formulas. Otherwise, they are α-related by the Definition (6.1.8) of a sequent. Therefore, $\zeta_P(\mathcal{I}(p'))$ and $pai^T_{\mathcal{I}}(p')$ are connectable. The conditions of the replacement rule are $\mathcal{POB}_P(\mathcal{I}(p'), pai^T_{\mathcal{I}}(p'))$. Therefore the application of the rule results in

$$\mathcal{POB}_P(\mathcal{I}(p'), pai^T_{\mathcal{I}}(p')) \wedge true \tag{6.81}$$

which can be simplified by means of the simplification rule (see Definition (5.2.19)) to

$$\mathcal{POB}_P(\mathcal{I}(p'), pai^T_{\mathcal{I}}(p')) \tag{6.82}$$

□

Lemma 6.4.10. *The modified task proof state is consistent.*

Proof. We have to show that the modified task is still a sequent. There are two cases to consider, depending on whether the replaced conclusion formula is a sequent formula or not.

Case 1: If the replaced conclusion is not a sequent formula, the substructures do not change during the application. As the task proof state was consistent before the application of the rule, the sequential property does also hold after the replacement of the substructure.

Case 2: If the replaced conclusion is a sequent formula, f' is constructed according to (6.65)/(6.66), respectively.

In both situations, the new windows are spanning for $pai^T_{\mathcal{I}}(c)$. Therefore, the combination of f,f' is spanning. Moreover, as all new windows are α-related to each other, the task state is sequential. □

Theorem 6.4.11 (Correctness of Backward Rule Application)**.** *Let $\mathcal{I}$ be an inference, T be a task and $pai^T_{\mathcal{I}}$ be a backward PAI. Then $pai^T_{\mathcal{I}}$ is admissible in the* Core *calculus. Moreover, the resulting proof state is consistent.*

Proof. By Lemma (6.4.8), Lemma (6.4.9), and Lemma (6.4.10) □

Discussion: We have shown that how assertion applications can conveniently modeled within the Core framework by a sequence of resolution replacement rules. Due to the imposed structure of the indexed task tree, the effect of the transformation can be determined locally by analyzing the inference and the task only. However, there is the question of whether the particular choice of this sequence is optimal with respect to the size of the induced proof obligations. This is because each instantiated formal argument of the inference induces two resolution replacement rules. In particular, we could also apply the resolution replacement rules induced by the premises first (from right to left) – intuitively this would correspond to a specialization of the inference –, and then apply the resolution replacement rule induced by the instantiated conclusion.

In the case that all premises are instantiated in other windows than the conclusion, both transformation yield the same result. This illustrates that our mechanism removes some of the redundancy of the search space. In the case that a premise is matched inside the window of a conclusion, we can show that the proof obligations introduced by our method are smaller than in the second possibility mentioned above. Intuitively, this is because we only have to collect once the expensive proof obligations arising from the conclusion resolution replacement rule that connects the inference part and the task part. To see that this is advantageous, consider the following task, where the formula Q is used as premise for the goal R:

$$\vdash P^+ \wedge \left(\boxed{Q^-} \Rightarrow \boxed{R^+}\right)^+ \tag{6.83}$$

Applying the premise resolution rule first induces a proof obligation $\neg P$, which is omitted in our case.

6.4.1 Generation of New Premises and Task Splitting

So far, the proof state is always represented by a single task together with its context. This is in contrast to the standard sequent calculus, where the so-called β-rules, such as $\wedge_R$, introduce several goals that need to be solved simultaneously[2]. While our formulation does not require such a splitting, we want to be able to model this situation within our approach, as it might have advantages for the interactive setting: When splitting a composed goal, the individual subgoals are smaller than the composed goal, and their contexts can locally be modified. Such a situation is in particular beneficial in the case that both subgoals do not share meta-variables. In this case, both goals can be solved independently from each other. Let us consider a simple example:

Example 6.4.12. *The application of set extensionality to the task*

$$A \cup B = B \cup A \tag{6.84}$$

results in the new task

$$\vdash A \cup B \subset B \cup A \wedge B \cup A \subset A \cup B \tag{6.85}$$

However, it would be desirable to obtain two tasks instead:

$$\vdash A \cup B \subset B \cup A \tag{6.86}$$

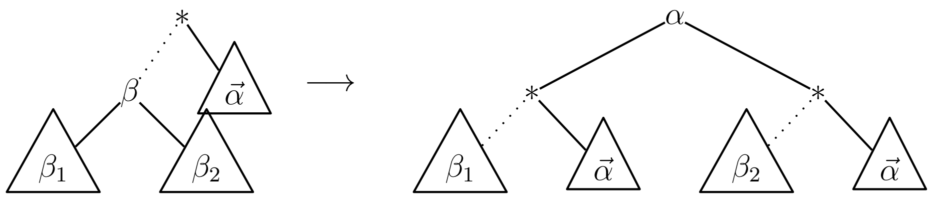
$$\vdash B \cup A \subset A \cup B \tag{6.87}$$

Within our framework, the main difference between (6.85) and (6.86) is that the window structure covering (6.85) is replaced by two window structures resulting in the two tasks shown in (6.86). It is this replacement which introduces a new sequent.

However, when performing this operation, we have to guarantee that each sequent satisfies the required properties of Definition 6.1.8 and that the sequents are sequential (see Definition 6.1.10). Particularly, in the presence of assumptions, subsequent operations might be necessary, as replacing a single window for a formula $A \wedge B$ by two windows for A, respectively B, introduces two windows that are β-related to each other, contradicting the requirement of Definition 6.1.8.

The above problem can be solved by reallocating certain nodes in the free task tree, as shown below:

[2]Note that substitutions can still have a global effect in the presence of meta-variables



This corresponds exactly to the application of Schütte's decomposition rule (see Section 5.2.7), which is admissible in the CORE calculus. Of course, for n subgoals, resulting from n premises, we can apply the Schütte rule $n-1$ times. Note that the rule also copies all the windows in the appropriate manner in order to obtain the two new sequents.

Definition 6.4.13 (Backward Inference Rule Application, Splitting)**.** *Let $\mathcal{I}$ be an inference with n premises, T be a task of the task tree (S, f), and $pai_{\mathcal{I}}^T$ be a PAI with respect to $\mathcal{I}$ and T, where k premises are instantiated. The effect of applying the PAI consists of replacing the task by n new subtasks $T_1', \ldots, T_n'$, where each subtask has the form*

$$T|_{pai_{\mathcal{I}}^T(c) \leftarrow \mathcal{POB}_{bw}(p_i)} \tag{6.88}$$

Theorem 6.4.14. *The backward application splitting rule is sound. The proof state resulting from the application of the backward splitting rule is consistent.*

Proof. As Theorem (6.4.11), then apply Schütte's decomposition rule $n-1$ times. □

6.5 Assertions: Forward Application

In the case of forward application, none of the conclusions of the inference has been instantiated, but at least for one premise p it holds that $pai_{\mathcal{I}}(p) \neq \perp$. The expected effect of the application of such an inference – at least when applied at top-level – is the derivation of a new fact, given by the conclusion of the inference, together with additional subgoals for the uninstantiated premises. While our inference mechanism will be much more general, it is our design goal to obtain the usual transformation described above when restricting the premise positions to top-level formulas in the antecedent of the sequent.

Similar to the case of backward application, the correctness of the transformation will be modeled by a sequence of resolution replacement rules, and as before, proof obligations arise due to uninstantiated premises or because of the deep access to subformulas. One minor difference is the fact that we do not want a premise to be removed during the application. We therefore distinguish one special premise among the instantiated premises, called *major premise*, which takes over the role of the conclusion and which is copied by the application of the contraction rule. The remaining premises are called *minor premises*. Before considering the technical details of the transformation, let us illustrate the transformation by means of an example.

Example 6.5.1. *We consider the inference*

$$\frac{p_1 : A \subset B \quad p_2 : x \in A}{c : x \in B}\,\textsc{Def-}\subset \tag{6.89}$$

and the task

$$S \subset T \Rightarrow \mathcal{P}(S) \subset \mathcal{P}(T), x \in \mathcal{P}(S) \vdash G \tag{6.90}$$

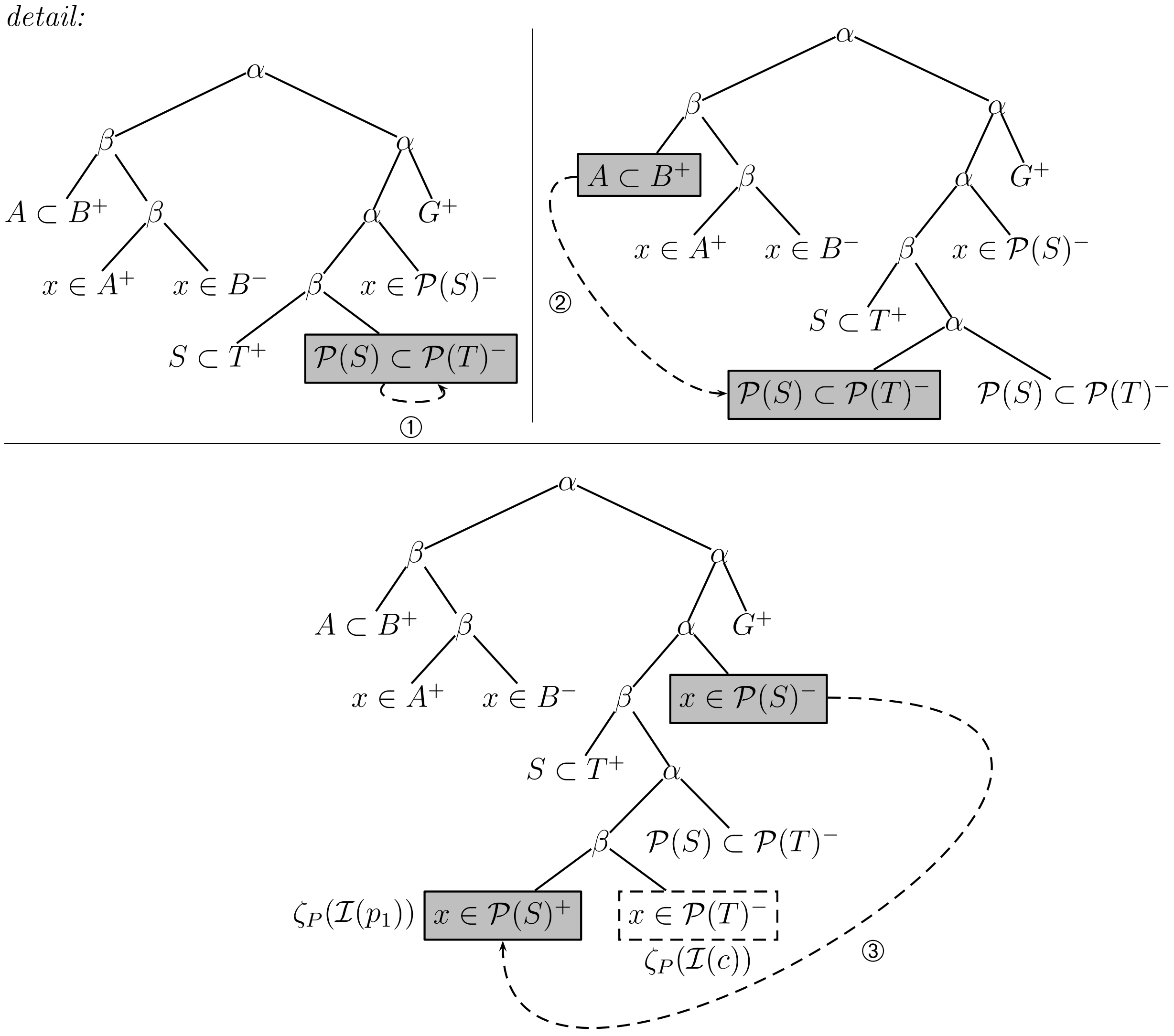
An application of the inference Def-*subset with* p_1 *instantiated with* $\mathcal{P}(S) \subset \mathcal{P}(T)$ *and* p_2 *instantiated with* $x \in \mathcal{P}(S)$ *transforms the task in one step to*

$$S \subset T \Rightarrow (x \in \mathcal{P}(T) \wedge \mathcal{P}(S) \subset \mathcal{P}(T)), x \in \mathcal{P}(S) \vdash G \tag{6.91}$$

provided that p_1 *is the major premise. In the case that* p_2 *is the major premise, the result is*

$$S \subset T \Rightarrow \mathcal{P}(S) \subset \mathcal{P}(T), S \subset T \Rightarrow x \in \mathcal{P}(T), x \in \mathcal{P}(S) \vdash G \tag{6.92}$$

Note that because of the different insertion position, the condition $S \subset T$ *is copied. We focus on the first possibility and show the sequence resulting in the transformation in detail:*



First, the major premise is contracted, as indicated by ①. In a second step (②), the replacement rule induced by the major premise is applied. As shown above, all premise nodes of the inference – except the major premise – as well as all conclusions (dashed box) have been transferred to the task part of the free variable task tree. As in the case of backward application, this induces a mapping ζ_P *that identifies a substructure of the replaced premise for each minor premise of the inference and all conclusions. This mapping is then used to apply all replacement rules induced by the minor premises, as indicated by ③. For convenience, it is useful to denote the left copy of the major premise* p *(after the application of the contraction rule) by* $\zeta_P(pai_{\mathcal{I}}^{T}(p))$.

For arbitrary inferences, the process depicted above can be generalized as follows:

1. Increase the multiplicity of all involved meta-variables of the inference.
2. Apply the contraction rule to the instantiated major premise, i.e., to $pai^T_{\mathcal{I}}(p)$. Denote the left copy by $\zeta_P(pai^T_{\mathcal{I}}(p))$.
3. Apply the resolution replacement rule induced by the major premise p of the inference, i.e., the replacement rule between the nodes
$$\mathcal{I}(p), \zeta_P(pai^T_{\mathcal{I}}(p)) \tag{6.93}$$
where $\zeta_P(pai^T_{\mathcal{I}}(p))$ is replaced. This results in the transfer of all minor premises $p_1, \dots, p_n$ as well as the conclusions $c_1, \dots, c_n$ of the inference to $\zeta_P(pai^T_{\mathcal{I}}(p)$. Denote them by $\zeta_P(\mathcal{I}(\overline{p_1})), \dots, \zeta_P(\mathcal{I}(\overline{p_n}))$[3] and $\zeta_P(\mathcal{I}(c))$ (for the combined conclusions).
4. For each instantiated minor premise p_i ($1 \le i \le n$) apply its induced resolution replacement rule, i.e., the replacement rule between the nodes
$$pai^T_{\mathcal{I}}(p_i), \zeta_P(\mathcal{I}(\overline{p_i})) \tag{6.94}$$
where $\zeta_P(\mathcal{I}(p'))$ is replaced.
5. Weaken the formula to clean the task tree.

As in the backward case, we can take advantage of the structural properties of task trees and define the proof obligation of a single premise locally. The overall proof obligations are then composed by the proof obligations of the minor premises.

Definition 6.5.2 (Forward Inference Conditions)**.** *Let $\mathcal{I}$ be an inference with premise labels $p_1, \dots, p_n$, T be a task, and $pai^T_{\mathcal{I}}$ be a forward partial argument instantiation with respect to $\mathcal{I}$ and T. For each label l denoting a minor premise, the conditions of l are then defined as follows:*
$$\mathcal{POB}_{fw}(l) = \begin{cases} \textbf{\textit{Label}}(\mathcal{I}(\overline{l}))\sigma & \text{if } pai^T_{\mathcal{I}}(l) = \bot \\ \mathcal{POB}_P(pai^T_{\mathcal{I}}(l), pai^T_{\mathcal{I}}(p))\sigma & \text{else} \end{cases} \tag{6.95}$$
The overall conditions of the $pai^T_{\mathcal{I}}$ are then given by
$$\bigwedge_{i=1}^{n} \mathcal{POB}_{fw}(p_i) \tag{6.96}$$
Therefore, the positions $pai^T_{\mathcal{I}}$ do not change.

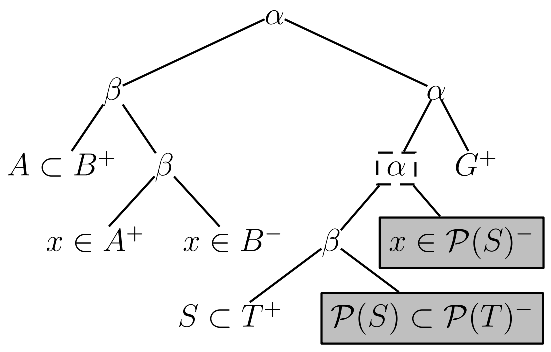
Remark 6.5.3. *(i) In Definition 6.5.2, the proof obligations of an instantiated minor premise l are defined with respect to the major premise p as $\mathcal{POB}_P(pai^T_{\mathcal{I}}(l), pai^T_{\mathcal{I}}(p))$. The arising conditions are exactly the same as $\mathcal{POB}_P(pai^T_{\mathcal{I}}(l), \zeta_P(pai^T_{\mathcal{I}}(p))$, but the former definition is more elegant as it avoids the use of ζ_P.*

(ii) The application of the contraction rule to the task position corresponding to the major premise $pai^T_{\mathcal{I}}(p)$ does not change the other positions, as task positions are non-overlapping (see Definition 6.3.5).

[3]As in the backward case, we use the notion $\overline{p}$ to denote "closure" of a premise, i.e., the premise together with its hypotheses, see Notation 6.2.8

To clarify the definition, we illustrate it by means of an example.

Example 6.5.4. *In Example 6.5.1 with p_1 as major premise, there is only one minor premise, namely p_2, which is also instantiated. Therefore, $\mathcal{POB}_{fw}(p_2)$ is given by the proof obligations between the task position of p_1, i.e., $pai^T_{\mathcal{I}}(p_1)$ and $pai^T_{\mathcal{I}}(p_2)$. The node that governs both positions is the one indicated by the dashed box; let us call the node a. As there are no β formulas on the path from $pai^T_{\mathcal{I}}(p_2)$ to a, there are no proof obligations, i.e., $\mathcal{POB}_{fw}(p_2) = \text{true}$.*



The overall effect of the forward application can now be defined as the replacement of the left copy of the major premise, i.e., $\zeta_P(pai^T_{\mathcal{I}}(p))$, by the overall proof obligation (see Definition 6.5.2). The windows are adapted as follows: If the major premise does not denote a top-level formula, that is, it does not correspond to a substructure corresponding to a window, nothing needs to be changed: There is a parent node covered by a window, and the parent node will still be covered after the replacement. As the window structure was spanning before, it will be spanning after the replacement. In the other case, where the major premise denotes a top-level formula, the windows need to be adapted: The original window covering the major premise is removed, and two new windows are inserted, one corresponding to the new fact, and one corresponding to the copy of the replaced fact.

Definition 6.5.5 (Forward Inference Rule Application)**.** *Let $\mathcal{I}$ be an inference with major premise p and minor premises $p_1, \dots, p_n$, T be a task of the task tree (S, f), and $pai^T_{\mathcal{I}}$ be a forward PAI with respect to $\mathcal{I}$ and T. Moreover, let c denote the node that is maximal with respect to $\prec$ and contains all conclusions. The effect of applying the PAI consists of replacing the task position corresponding to the major premise by a copy of it and the new fact including its induced proof obligations, i.e.,*

$$S|_{pai^T_{\mathcal{I}}(p) \leftarrow (S', f')} \tag{6.97}$$

where S' is the substructure of the formula tree corresponding to the formula

$$\alpha(\beta(\bigwedge_{i=1}^{n} \mathcal{POB}_{fw}(p_i), \boldsymbol{Label}(\mathcal{I}(c)), \boldsymbol{Label}(pai^T_{\mathcal{I}}(p)) \tag{6.98}$$

and f' is defined as follows:

- *If $pai^T_{\mathcal{I}}(p)$ is an antecedent formula, i.e., there is an $m \in dom(f)$ with $f(m) = pai^T_{\mathcal{I}}(p)$,*

$$f'(n) := \begin{cases} s_1 & \text{if } n = m \\ s_2 & \text{if } n = m' \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6.99}$$

  where s_1 and s_2 denote the immediate substructures of $f(m)$.

- *Otherwise $f'(n)$ undefined for all $n \in dom(f')$.*

Example 6.5.6. *The overall effect of a forward application of an inference has already been illustrated in Example 6.5.1. In the example, the node corresponding to $\mathcal{P}(S) \subset \mathcal{P}(T)$ is replaced by a new formula tree, which is composed of the following parts*

- *the proof obligations from p_2, i.e.,* true.
- *the conclusion of the inference, i.e., $x \in \mathcal{P}(T) = \textbf{\textit{Label}}(\mathcal{I}(c))$.*
- *a copy of the major premise, i.e., $\mathcal{P}(S) \subset \mathcal{P}(T) = \textbf{\textit{Label}}(pai_{\mathcal{I}}^{T}(p))$.*

The window structure is not changed, as there is no window pointing to the node $\mathcal{P}(S) \subset \mathcal{P}(T)$.

We show subsequently that the effect of the application of $pai_{\mathcal{I}}^{T}$ can be modeled by a sequence of resolution replacement rules. We proceed stepwise and characterize first the structure of the task tree after the application of the contraction rule and the replacement rule induced by the major premise.

Theorem 6.5.7. *Let $\mathcal{I}$ be an inference with minor premises $p_1, \ldots, p_n$ and major premise p. Moreover, let c denote the inference node that is maximal with respect to $\prec$ and contains all conclusions, T be a task and $pai_{\mathcal{I}}^{T}$ be a forward PAI. After the application of the contraction rule to $pai_{\mathcal{I}}^{T}(p)$, the replacement rule induced by the major premise $pai_{\mathcal{I}}^{T}$ is admissible. The overall replacement of both steps is*

$$pai_{\mathcal{I}}^{T}(p) \to \alpha(\beta(\bigwedge_{i=1}^{n} \textbf{\textit{Label}}(\mathcal{I}(\overline{p_i})), \textbf{\textit{Label}}(\mathcal{I}(c))), \textbf{\textit{Label}}(pai_{\mathcal{I}}^{T}(p)))\sigma \tag{6.100}$$

Before giving the proof of this theorem, let us clarify the involved notation by means of an example.

Example 6.5.8. *Let us again consider Example 6.5.1. After the application of the contraction rule, we are in the situation depicted top right in the picture on page 102. Application of the replacement rule induced by the major premise results in the β-insertion of all nodes on the path to the root node that are maximal and β-related to $A \subset B$. In the example, this is the single node $x \in A \Rightarrow x \in B$. The theorem characterizes this proof obligation on a finer level, namely as conjunction of all minor premises (including their hypotheses), i.e., $x \in A$ for our example, and the conclusion $x \in B$ in our example. Indeed, $\beta(x \in A^{+}, x \in B^{-}) = x \in A \Rightarrow x \in B$.*

Proof. Let R_c denote the node in the task tree corresponding to $pai_{\mathcal{I}}^{T}(p)$. According to Definition 5.2.16, the contraction rule is applicable and results in the replacement of R_c by

$$\begin{tikzcd}[column sep=small] & \alpha(\mathbf{Label}(R_c), \mathbf{Label}(R'_c)) \arrow[dl, dash] \arrow[dr, dash] & \\ R_c & & R'_c \end{tikzcd} \tag{6.101}$$

where R'_c is a copy of R_c. Due to condition (iii) of Definition 6.3.5, the task positions of the minor premises are not affected. We subsequently work on R_c to perform the replacement. The replacement rule induced by p is between the R_c and $\mathcal{I}(p)$. Note that $\mathcal{I}(p)$ has positive polarity, while R_c inherits the polarity of $pai_{\mathcal{I}}^{T}(p)$, which is negative. Moreover, both nodes are α-related via the root node of the task tree due to Theorem 6.1.12 and equal under

σ. Therefore, the replacement rule is admissible. Let $n_1, \ldots, n_m$ denote the maximal nodes from $\mathcal{I}(p)$ to the root node that are β-related to $pai^T_{\mathcal{I}}(p)$. The application of the PAI results in β-insertion of $n_1, \ldots, n_m$. According to Theorem 6.2.5, the premises are β-related to each other and β-related to all conclusions and all nodes covered. Therefore, $\beta(n_1, \ldots, n_m) = \beta(p_1, \ldots, p_n, c)$. Consequently, R_c is replaced by

$$\left(\beta(\bigwedge_{i=1}^{n} \mathbf{Label}(\mathcal{I}(\overline{p_i})), \mathbf{Label}(c)) \wedge \text{true}\right) \sigma \tag{6.102}$$

which, by the application of the simplification rule (see Definition 5.2.19) is simplified to

$$\beta(\bigwedge_{i=1}^{n} \mathbf{Label}(\mathcal{I}(\overline{p_i})), \mathbf{Label}(c))\sigma \tag{6.103}$$

Consequently, the overall replacement is (6.100). □

We now show that replacement rules induced by the minor premises are admissible and result in the proof state as specified in Definition 6.5.5. As a result, each instantiated minor premise is replaced by their induced proof obligations.

Example 6.5.9. *Let us again consider Example 6.5.1. After the application of the contraction rule and the replacement rule induced by the major premise, we are in the situation depicted at the bottom in the picture of page 102. Application of the induced replacement rule for p_2 results in the replacement of $x \in \mathcal{P}(S)$ by* true.

Theorem 6.5.10. *All induced premise resolution replacement rules are admissible. Application of all induced premise resolution replacement rules transforms* (6.100) *to the conditions of* (6.98)*, i.e.,*

$$\alpha(\beta(\bigwedge_{i=1}^{n} \boldsymbol{Label}(\mathcal{I}(\overline{p_i})), \boldsymbol{Label}(c)), \boldsymbol{Label}(pai^T_{\mathcal{I}}(p)))\sigma$$
$$\rightarrow \alpha(\beta(\bigwedge_{i=1}^{n} \mathcal{POB}_{fw}(p_i), \boldsymbol{Label}(c)), \boldsymbol{Label}(pai^T_{\mathcal{I}}(p)))\sigma \tag{6.104}$$

Proof. The proof is done by induction over the number of instantiated minor premises $|inst(P)|$. It is sufficient to consider the replacement

$$\bigwedge_{i=1}^{n} \mathbf{Label}(\mathcal{I}(\overline{p_i})) \rightarrow (\bigwedge_{i=1}^{n} \mathcal{POB}_{\text{fw}}(p_i)) \tag{6.105}$$

as all replacements occur within $\mathbf{Label}(\mathcal{I}(\overline{p_i}))$.

Base Case: If $|inst(P)| = 0$, then $\mathcal{POB}_{\text{fw}}(p_i) = \mathbf{Label}(\mathcal{I}(\overline{p_i}))$ and (6.100) has already the desired form.

Step Case: Let us now assume that $|inst(P)| = n$. Let $p_1, \ldots, p_{n-1}$ denote the first $n-1$ minor premises. By induction hypothesis, we can assume that the first $n-1$ replacement rules are admissible. Therefore, we have

$$\bigwedge_{i=1}^{n} \mathbf{Label}(\mathcal{I}(\overline{p_i})) = \bigwedge_{i=1}^{n-1} \mathbf{Label}(\mathcal{I}(\overline{p_i})) \wedge p_n \tag{6.106}$$
$$\rightarrow (\bigwedge_{i=1}^{n-1} \mathcal{POB}_{\text{fw}}(p_i)) \wedge p_n \tag{6.107}$$

p_n induces a resolution replacement rule between $\zeta_P(\mathcal{I}(p_n))$ and $pai^T_{\mathcal{I}}(p_n)$. Both have opposite polarity. Moreover, they are α-related due to Definition 6.3.5 and therefore connectable via the substitution σ. The conditions of the replacement rule are $\mathcal{POB}_P(\zeta_P(\mathcal{I}(p_n)), pai^T_{\mathcal{I}}(p_n))$. Therefore the application of the rule results in

$$(\bigwedge_{i=1}^{n-1} \mathcal{POB}_{\text{fw}}(p_i)) \wedge \mathcal{POB}_P(\zeta_P(\mathcal{I}(p_n)), pai^T_{\mathcal{I}}(p_n)) = \bigwedge_{i=1}^{n} \mathcal{POB}_{\text{fw}}(p_i) \tag{6.108}$$

□

What remains to be shown is that the resulting task proof state is consistent.

Lemma 6.5.11. *The modified task proof state is consistent.*

Proof. We have to show that the modified task is still a sequent. There are two cases to consider, depending on whether the replaced conclusion formula is a sequent formula or not.

Case 1: If the replaced conclusion is not a sequent formula, the substructures do not change during the application. As the task proof state was consistent before the application of the rule, the sequential property does also hold after the replacement of the substructure.

Case 2: If the replaced conclusion is a sequent formula, f' is constructed according to (6.99) respectively.

In both situations, the new windows are spanning for $pai^T_{\mathcal{I}}(c)$. Therefore, the combination of f,f' is spanning. Moreover, as all new windows are α-related to each other, the task state is sequential. □

Putting our observations together, we obtain the correctness result of the forward application.

Theorem 6.5.12 (Correctness of Forward Rule Application)**.** *Let $\mathcal{I}$ be an inference, T be a task and $pai^T_{\mathcal{I}}$ be a forward PAI. Then the effect of $pai^T_{\mathcal{I}}$ can be modeled by a sequence of* CORE *inference rules. Moreover, the resulting proof state is consistent.*

Proof. By Theorem (6.5.7), Theorem (6.5.10), and Lemma (6.5.11) □

Discussion: The transformation defined above contains an explicit choice point, namely the selection of the major premise (see Example 6.5.1), upon which the resulting proof state depends. The choice point emerges as soon as more than one premise of a given inference is instantiated, because the only condition the major premise has to satisfy is that the corresponding premise is instantiated in the PAI. Therefore, one may ask whether and to what extent the choice matters. To answer this, one must understand that the major premise specifies the position at which the new fact, including the condition upon which it depends, is inserted. It is important to note that the conditions themselves depend on this position and might be reduced in the case that several premises are matched within the same sequent formula. As an example, consider the following situation:

$$A \vdash P \wedge (Q \Rightarrow R) \tag{6.109}$$

and imagine an inference that allows the derivation of the fact B given the facts A and Q. Selecting A as major premise results in the task

$$A, \neg P \Rightarrow B \vdash P \wedge (Q \Rightarrow R) \tag{6.110}$$

whereas using Q as major premise results in

$$A \vdash P \wedge ((Q \wedge B) \Rightarrow R) \tag{6.111}$$

Intuitively, in the second case the new fact is locally derived for the subgoal R, while in the former case it is derived for all available goals. As a consequence, the condition $\neg P$ needs not to be introduced.

Despite the presence of situations as illustrated above, let us note that it is common in practice to restrict the candidates for premises to those subformulas which do not introduce proof obligations. In this case, the choice of the major premise does not matter.

6.6 Application of Rewrite Rules

Application of rewrite rules is directly supported by the Core framework using rewrite replacement rules (see Section 5.2.1). Each ϵ-node, i.e., negative node of the form $s = t$ or $s \Leftrightarrow t$, gives rise to a rewrite replacement rule. Rewriting can be performed in two directions: forwards, that is, replacing an instance of s by t, or backwards, that is, replacing an instance of t by s. In contrast to resolution replacement rules, rewrite replacement rules can also be applied inside literal nodes to modify a subterm of a literal node. We will subsequently define an abstract interface for rewriting with equality and equivalences.

Within Core, the application of rewrite replacement rules splits into two cases, depending on whether the replacement occurs inside a literal node or whether the complete node is replaced. In case an equation or equivalence $\epsilon(s, t)$ is used to rewrite a node, Leibniz' equality introduction is used to obtain $s \Rightarrow t$ or $t \Rightarrow s$ of opposite polarity to the node that shall be rewritten. The case is then identical to the application of a normal inference as defined in the previous section, using the backward application in the case that the node to be rewritten has positive polarity, and forward application otherwise (with major premise $s \Rightarrow t$).

In case that the replacement occurs inside of a literal node, the rewrite replacement rule is modeled as a sequence of extensionality introduction, Leibniz equality, and a resolution replacement rule, as illustrated in Example 5.2.15 on page 61. As for the case of rewriting nodes, the preparatory Core rules construct a formula of the form $\varphi[s] \Rightarrow \varphi[t]$, which is then finally applied by means of a resolution replacement rule. Therefore, this case can also be handled by the previously introduced mechanism for inferences. Of course, the rewrite rule is only applicable in the case that the preparatory steps can be executed.

Remark 6.6.1. *If no binders are involved, this is always possible. In the case that a term below a λ-binder is to be rewritten, the extensionality introduction rule might fail. Let π denote the label of the literal node which shall be rewritten at position π and let $x_1, \ldots, x_n$ denote the variables that are free in $\varphi_{|\pi}$ but not in φ (these are exactly the variables that are captured by a λ). Let σ' be the substitution for which $\sigma'(s) = \sigma'(\varphi_{|\pi})$. First, σ' must not capture bound variables, i.e., $x_i \notin dom(\sigma')$ for $1 \leq i \leq n$. Second, the variables within the domain of σ' that are related to the bound variables, i.e., those $x \in dom(\sigma')$ for which $x_i \in \sigma'(x)$ for some i, must be γ-local in order for the extensionality rule to* *be applicable (see Definition 5.1.14). This means that no condition involves one of the variable (see Definition 5.1.13). This is intuitively clear, as we cannot, for example, verify the condition $x \neq 0$ within the term $\lambda x.(x+1)(x+1)^{-1}$. If desired, a possible work-around is the introduction of a if-then-else construct, which puts the condition inside the binder (see [Aut03] p. 88 for a discussion), but so far we have not experienced a case where this is desired.*

6.7 Related Work

The assertion application framework developed within this chapter provides the possibility for sound reasoning with assertions in a sequent style, combined with the feature that the assertions can be applied deeply inside formulas. The application of an assertion can be seen as a combination of several deduction steps into one, which is the common feature of so-called *hyper calculi* (see [Häh01]). This has usually the following two advantages: (i) proof search becomes faster, and more importantly (ii) some intermediate results are not computed in the first place, leading to a considerably smaller search space.

As the CORE calculus on which our work is based is relatively new and not widely known yet, we compare in this section the assertion mechanism with developments in the sequent calculus and natural deduction. Let us note that the algorithm that computes the inference representation (Section 6.2) can easily be transferred to the sequent calculus or natural deduction: Each window transition corresponds to a decomposition rule and the computation of an inference to the computation of its normal form. However, the deep inference feature requires additional work.

6.7.1 LeanT^AP

LeanT^AP [BP95, Fit97] implements a complete and sound theorem prover for classical first-order logic based on free-variable semantic tableaux. It follows the paradigm of *lean deduction*, resulting in very small, yet efficient PROLOG implementation that consists only of a few lines of code. The ideas of LeanT^AP have been the starting point to automate proof search in ISABELLE, as described in [Pau99]. In a nutshell, LeanT^AP is based on the following principles:

- Given a formula to be proved, it is preprocessed to a Skolemized negation normal form. As a result, the number of cases that need to be considered in the main loop are reduced.
- For each branch, LeanT^AP maintains a current formula to be expanded, a queue of yet unexpanded formulas, and a list of literals that have so far been discovered on the branch.
- A branch can be closed if two complementary literals are found on the branch. In this case, the closing substitution is applied to the other branches. However, even if a branch can be closed, the alternatives to close that branch have to be considered further for completeness.
- A formula is completely expanded on a branch. The underlying idea is to produce literals to close a branch as fast as possible. For γ-formulas, a single instance is generated and further expanded, but the original γ-formula is put at the end of the

list of unexpanded literals. This guarantees a fair treatment of the formulas on a branch and that sufficiently many instances are generated.

- To obtain a complete method, an additional variable `VarLim` restricts the number of allowed instances of γ-variables on a branch. If no proof could be found within the limit `VarLim`, `VarLim` is increased and the proof search started again.

Even though LeanT^AP does not derive new inference rules during its processing, its proof strategy comes already very close to the application of assertions, as an assertion is fully decomposed once it has been selected on the current branch. However, intermediate facts resulting from a decomposition are not removed, but added to the set of unexpanded formulas. While this is unproblematic for α-formulas, for which the decomposition is just postponed, in the case of γ-formulas the number of formulas on a branch increases rather quickly, as each decomposition introduces a new copy. In particular, a formula $\forall x, y.P(x, y)$ introduces a copy of itself, and the decomposition a copy of $\forall y.P(X, y)$, which is not necessarily needed, as it can also be constructed from the original formula. Deep inference is not supported by LeanT^AP.

6.7.2 Focusing

Focusing, which was originally developed in the context of classical linear logic [And92], is a technique that removes inessential nondeterminism from the proof search. It is based on the observation that invertible rules can always be applied eagerly in any order in a backward proof without loosing completeness. For example, the goal $A \Rightarrow B$ can always be decomposed to prove B under the additional assumption A. Therefore, it is reasonable to chain together invertible rules. Interestingly, a similar result holds for non-invertible rules: As long as the top-level connective is not invertible, we can continue the decomposition of the formula, making a choice at each step.

To take advantage of this observation, formulas are refined to *polarized formulas*, where each formula has a polarity[4]. A connective is positive if its left rule is invertible, and negative, if its right rule is invertible. Note that while in linear logic the polarity of a connective is uniquely determined, this property does not hold in intuitionistic logic. Proof search is then organized in two phases: an *inversion phase* and a *focusing phase*, which are alternating. In the inversion phase, which is the first phase, all invertible rules are applied, both on the left and right, where a specific order is forced to avoid choice, even though the concrete order does not matter. This phase eventually ends with a so-called *neutral sequent*, and represents the switch to the focusing phase. The focusing phase selects a proposition (from the left or from the right) that obtains a focus. The focused proposition is then decomposed until it becomes atomic or changes its polarity, in which case the inversion phase is started again.

As mentioned above, the result of the inversion phase is a neutral sequent in which the next formula to be focused on has to be selected. Such a phase is called *block* or *dipole* in the literature and goes from a neutral sequent to neutral sequents. It can be used to derive big-step rules that go from stable sequents to stable sequents as follows: Given a proposition to be focused on, and call its subformulas that could appear in neutral sequents *syntactic connectives*. Then, each block can be considered as the application of a left or right rule for a synthetic connective. Up to now, deep inference has not been combined with the focusing approach.

[4]Note that polarized formulas differ from signed formulas

6.7.3 Prawitz, Supernatural Deduction, Superdeduction

The idea of extending the natural deduction calculus or the sequent calculus by new deduction rules is not new. In [Pra65] Prawitz already defines deduction rules that allow the replacement of atomic propositions P by the formula Q if the axiom $\forall \vec{x} P \Leftrightarrow Q$ exists in the current context and proves the admissibility of cuts[5], provided that P does not occur in Q. In the literature, these rules are known as *folding* and *unfolding* rules (called λ-introduction and λ-elimination by Prawitz). In the context of set theory, this can be used to replace the axiom $\forall x, y.(x \in \mathcal{P}(y) \Leftrightarrow \forall z(z \in x \Rightarrow z \in y)$ by two rules

$$\frac{\Gamma \vdash \forall z.(z \in x \Rightarrow z \in y)}{\Gamma \vdash x \in \mathcal{P}(y)}\ \text{FOLD} \qquad \frac{\Gamma \vdash x \in \mathcal{P}(y)}{\Gamma \vdash \forall z.(z \in x \Rightarrow z \in y)}\ \text{UNFOLD}$$

An overall discussion including links to investigations with respect to specific theories can be found in [Dow09].

The idea of Prawitz has been extended to supernatural deduction [Wac05b] and superdeduction [BHK07a], which allows the derivation of new rules from propositional rewrite rules, i.e., formulas of the form $\forall \vec{x}.P \Leftrightarrow Q$ where P is an atomic proposition and where Q is immediately decomposed (in contrast to Prawitz) and for which cuts are also admissible.

Both approaches are restricted to closed, universally quantified equations or equivalences and the premises and conclusions of the derived inference rules are restricted to atomic formulas. Moreover, there is no possibility to apply inference rules deeply inside. On the other hand, our focus so far was less on proof theoretic properties, such as cut elimination, which is an interesting topic for future work. A promising approach to obtain such a result would be to follow the ideas of [BK07].

6.7.4 Deduction Modulo

Rather than externalizing theory knowledge into new derived rules, deduction modulo [DHK98] allows the internalization of theory knowledge by integrating deduction with respect to some standard calculus like for example sequent calculus and term rewriting systems. It extends the calculus rules by an equivalence relation provided by a background theory to check the equality of formulas and terms during the application of the calculus rules. Like superdeduction, this approach is restricted to formulas that give rise to rewrite rules (though extensions exist to accommodate forward- and backward-chaining).

6.7.5 Relationship to Hyperresolution and SLD Resolution

Hyperresolution is a complete strategy for resolution employed in the *Otter* [Kal01] family of theorem provers. The general idea is to combine several resolution steps in one big step. Hyperresolution comes in two flavors, a positive and negative variant; we describe positive hyperresolution[6]. This is done by dividing the available clauses into two sets: (i) *Nucleii*, containing one ore more negative literals, and (ii) *electrons*, containing no negative literals (i.e., positive clause). Resolution occurs between one or more electrons and one nucleus, where one electron clause is used for each negative literal in the nucleus, resulting in

[5]The notion of a cut is extended in a way that a sequence of a fold rule followed by an unfold rule is considered to be a cut

[6]Positive hyperresolution is obtained by replacing negative by positive and vice versa in the subsequent presentation

a new electron. Thus, one hyperresolution step corresponds to several resolution steps. Similarly, *hyper tableau* [BFN96] is a form of clausal tableau in which all negative literals in a clause are resolved away in a single inference step.

Compared to hyperresolution, there are two similarities. First, a single inference application (in the sequent calculus) itself represents a hyper-step, as it combines several deduction rules in a single step. Second, given an inference, we can instantiate several premises resulting in several resolution replacement rules.

Note that the process of deriving an inference rule for an assertion A can be seen as a conversion of A to disjunctive normal form: When putting all literals in the antecedent, each branch corresponds to exactly one clause. However, in the special case that each branch contains only a single formula, the disjunction of all branches yields a single clause. This is the case if the assertion is for example of the form $A_1 \wedge \ldots \wedge A_m \Rightarrow C$, in which the result is a Horn clause $\{\neg A_1, \ldots, \neg A_m, C\}$. Hyperresolution would resolve each negative literal, that is, each A_i with a positive clause. Within our setting, an inference rule is always applied within a single branch, and the instances are all found within the sequent, which is not normal yet. However, the introduction of a negative literal can be understood as the introduction of a new proof obligation. Therefore, we can understand the process as the restriction of the available assumptions to those positions that do not introduce proof obligations.

We have seen in the case above that under specific circumstances an assertion is converted to a single clause. Let us consider n goals of the form $\vdash L_i$ which correspond to a single clause $\{\neg L_1, \ldots, \neg L_n\}$. The application of an inference in backward direction to one literal L_i replaces that goal by $A_1 \wedge \ldots \wedge A_m$, which, when fully decomposed, result in m new branches $A_1, \ldots, A_m$, i.e., the new clause $\neg L_1, \ldots, \neg L_{i-1}, \neg L_{i+1}, \ldots, \neg L_n, \neg A_1, \ldots, \neg A_n$. Consequently, we obtain a new clause where L_i has been replaced by $A_1, \ldots, A_m$, which is similar to SLD resolution.

6.7.6 Imps

The Imps system [FGT93a, FGT93b] is close to Prawitz' ideas. In Imps, so-called *macetes* are generated from definitions and theorems, which are special simplifiers to manipulate expressions by applying theorems. Given a formula $\forall x_1 : \sigma_1, \ldots, \forall x_n : \sigma_n.\varphi$, where φ is not a universal formula, conditional rewrite rules that are determined by the structure of φ are extracted. Macetes can be combined to compound macetes using a language similar to tacticals. Macetes only work with pure universal formulas in prenex normal form and require the formulas to be given in the right format; the conditions are not further decomposed and equations (and equivalences) are only applied from left to right. Moreover, they are used within the interactive setting and have not been investigated from a theoretical point of view.

6.7.7 Muscadet

Muscadet [Pas01] is a knowledge based theorem prover based on natural deduction that aims to imitate methods as used by humans. It is based on rules and metarules which are applied to facts and objects. Facts result from the initial theorem to be proved or are added by rules. The rules comprise natural deduction rules or are generated from definitions and theorems by metarules. An example of a logical rule is "If the conclusion is $H \Rightarrow C$, then add the hypothesis H and the new conclusion is C". However, instead of directly adding the hypothesis H, the hypothesis is further processed by splitting H

in the case that H is a conjunction. Metarules are rules that are used to construct new rules from facts. For example, the rule "If there are hypotheses $A \subset B$ and $X \in A$, then add the hypothesis $X \in B$" is automatically build from the definition of subset. Interestingly, no meta-variables are used, and the rule is restricted to situations where all facts are present in the proof state. Instead, the available objects are scanned to control the forward exploration: If there is an object $A \cup B$ and a hypothesis $x \in A$, then add the hypothesis $x \in A \cup B$. There is no backtracking. Rules cannot be applied deeply inside formulas.

In practice, MUSCADET performs very well in the domain of set theory (see [Pas02]). Similar to our approach, this is due to the goal-directed proof search based on generated inference rules. However, let us point out that there is no formal investigation of the underlying principles, and the output generated by the system is hardly readable.

6.7.8 Theorema

Similar to MUSCADET, THEOREMA [BJD98] maintains available facts in a knowledge base. Each formula of the knowledge base is thereby transformed into one or more rewrite rules, which are classified into two categories: Rewrite rules for goal reduction, and rewrite rules for knowledge expansion. It is only possible to generate a rewrite rule if the right-hand side of the rule does not contain free variables. Implicational rewrite rules, which are rewrite rules of the form $\forall \vec{x}.A \Rightarrow B$, can also be applied to subformulas based on a classification of the subformulas by polarities. There are no proof theoretical investigations of this approach.

Compared with THEOREMA, our inferences are more general, as they do not require the formulas to be of a specific form; in particular we do not require the absence of free variables in the right-hand side of the rule. While deep replacement is possible in THEOREMA, it is not possible to match several formulas deeply inside, such as the axiom rule which requires the instantiation of both its premise and its conclusion. It is exactly this rule which reduces the size of the proof (see Section 13).

6.8 Summary

In this chapter we defined an assertion-level interface on top of the CORE calculus which supports proof search directly at the assertion level (Contribution A1(i), Section 1.1). The general idea was to impose a certain structure and invariants on a CORE proof state such that it essentially looks like a sequent calculus without quantifiers together with transformation rules in the form of inferences which are automatically computed from the available assertions. This results in a familiar representation of the proof state while guaranteeing the correctness of the transformations and keeping the technical details hidden from the user. By mapping inference rule applications to transformations on the indexed formula tree, inference rules (and thus assertions) can not only be applied to top-level formulas as known from the sequent calculus, but also to subformulas (Contribution A1(iii), Section 1.1). In particular, by taking advantage of the additional structural properties of the proof state it became possible to define the application of an assertion locally to a sequent (a particular subtree of the indexed formula tree) rather than considering the complete indexed formula tree.

7 Proof Theory

In this chapter, we characterize assertion level proofs for the standard sequent calculus in first order logic and study their proof theoretic properties. Interestingly, we can see assertion level reasoning as a special search strategy of the sequent calculus, in which a selected formula is fully decomposed and intermediate copies that arise when instantiating γ-rules are discarded. This search strategy sees a decomposition of a formula as a macro operator and caches it for efficiency, resulting in the inference rules.

7.1 Formal Characterization of Assertion Applications

The main feature of the application of an assertion A is that A is completely decomposed and intermediate γ-formulas are not copied. In other words, the assertion is completely used. To formalize that observation, we introduce the concept of A-active derivations to denote those derivations where only rule applications with principal formula A or one of its subformulas are applied. Each assertion can be applied arbitrarily often; thus it is allowed to perform contraction steps on A.

Definition 7.1.1 (A-Active/Passive Derivations). *Let L, L' be multisets of formulas and D a derivation (possibly with open goals) for the sequent $\Gamma, L \vdash L', \Delta$. The derivation D is (L, L')-*active*, if it contains only calculus rules having a formula from L or L' or one of their subformulas as principal formula. Conversely, we say D is (L, L')-*passive* if it contains no calculus rule that has some formula from L or L' or one of their subformulas as principal formula. If $L = \{A\}$ and $L' = \emptyset$ (respectively if $L' = \{A\}$ and $L = \emptyset$) then we agree to say D is A-active if it is (L, L')-active and A-passive if it is (L, L')-passive.*

A derivation D that is A-active is called maximal*, if it cannot be extended to a derivation D' that is A-active.*

We illustrate the concept of being A-active by means of an example:

Example 7.1.2. *The following derivation is not A-active, where $A = (G \Rightarrow F) \Rightarrow H$. This is because on the right branch the rule $\Rightarrow_R$ is applied to a $F \Rightarrow H$, which is not a* *subformula of* $(G \Rightarrow f) \Rightarrow H$*.*

$$\Rightarrow_L (A)\ \dfrac{\Rightarrow_R (B)\ \dfrac{H, F \vdash H}{H \vdash \boxed{F \Rightarrow H}} \qquad \dfrac{F \vdash G \Rightarrow F, H}{\vdash G \Rightarrow F, \boxed{F \Rightarrow H}}\ \Rightarrow_R (B)}{\underbrace{\boxed{(G \Rightarrow F) \Rightarrow H}}_{A} \vdash \underbrace{F \Rightarrow H}_{B}}$$

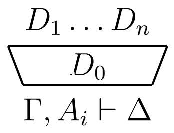
However, the following derivation is A-active:

$$\Rightarrow_L (A)\ \dfrac{H \vdash F \Rightarrow H \qquad \dfrac{G \vdash F, F \Rightarrow H}{\vdash \boxed{G \Rightarrow F}, F \Rightarrow H}\ \Rightarrow_R (A)}{\underbrace{\boxed{(G \Rightarrow F) \Rightarrow H}}_{A} \vdash \underbrace{F \Rightarrow H}_{B}}$$

This allows us to classify those derivations in the sequent calculus which have this special form as *assertion level derivations*.

Definition 7.1.3 (Assertion Level Derivation)**.** *Let* $A_1, \ldots, A_n$ *be assertions and* Π *be a derivation of* $\Gamma, A_1, \ldots, A_n \vdash \Delta$*.* Π *is said to be an* assertion level derivation *if the following conditions hold:*

- *No contraction step is applied to a subformula of one of the* A_i*.*
- *Every rule application on one of the* A_i *is of the form*



where the D_0 *is* A_i*-active and* $D_1, \ldots, D_n$ *are* A_i*-passive.*

It is called atomic assertion level derivation *if* D_0 *is maximal.*

7.2 Soundness and Completeness

As assertion level derivations are just particular sequent calculus derivations, it is clear that assertion level derivations are sound. However, as some restrictions are imposed on the structure of the derivations, it is natural to ask whether such derivations do exist in general, and if not, whether there are subclasses for which they do exist. This question will be explored subsequently.

There cannot be a decision procedure for first-order logic. This is because of the γ-rules which allow the introduction of an arbitrary *witness term* t. To simplify the subsequent completeness proof, we will make the following simplifications:

Deterministic Expansion Strategy: To obtain a simple, yet complete method, our strategy will be to make the expansion of formulas deterministic, that is, (i) to determine which formula to expand next (ii) to determine the instance to be used in the application of a γ-rule.

Work with Block Tableaux: Instead of working with sequents, it is technically easier to work just with sequences of signed formulas. That is, a sequent $\gamma_1, \ldots, \gamma_n \vdash \delta_1, \ldots, \delta_m$ will be represented as the sequence $T\gamma_1, \ldots, T\gamma_n, F\delta_1 , \ldots, F\delta_m$. Therefore, we will switch to tableau systems, more specifically to block tableaus, which directly simulate the sequent calculus and vice versa. We then show a completeness proof for systematic block tableaus, which is then transferred to systematic assertion level block tableaus.

Skolemize away Existential Quantifiers: We assume all formulas to be in Skolem normal form, that is, all existential quantifiers have been removed from the formula. This is not a serious restriction, as such a form can be computed for each formula automatically as a preprocessing step. Moreover, if desired, we can easily add further rules to our tableau procedure to perform skolemization at runtime.

7.2.1 Sequent Calculus and Block Tableau Systems

In an analytic tableau, each node of the tableau is labelled with a single formula. In contrast, in a block tableau a node is labelled with a set of formulas, corresponding to the proof nodes of an analytic tableau which still need to be considered for expansion. Therefore, a block tableau can be simulated by an analytic tableau and vice versa. Moreover, a block tableau can be seen to be just a different notation for a Gentzen system, as illustrated in Table 7.1. Note however that it is common to view the tableau method as a proof by contradiction, while the sequent calculus is understood to be a direct proof method. Figure 7.1 shows both an analytic tableau and a block tableau.

Remark 7.2.1. *It is common for tableau to grow downwards, while sequents usually grow upwards. Moreover, for tableau systems we separate different extensions (and-branching) by a vertical bar.*

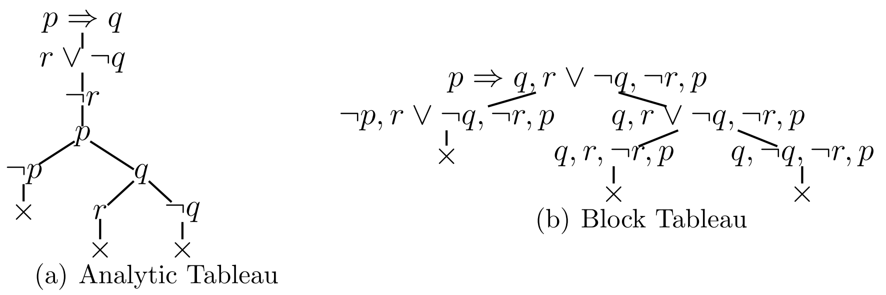

$p \Rightarrow q$
$r \vee \neg q$
$\neg r$
p
$\neg p$
q
r
$\neg q$
$p \Rightarrow q, r \vee \neg q, \neg r, p$
$\neg p, r \vee \neg q, \neg r, p$
$q, r \vee \neg q, \neg r, p$
$q, r, \neg r, p$
$q, \neg q, \neg r, p$
(a) Analytic Tableau (b) Block Tableau

Figure 7.1: Analytic vs. Block tableau for $p \Rightarrow q, r \vee \neg q, \neg r, p$

7.2.2 Systematic Block Tableau

In this section, we introduce a systematic block tableau and show its completeness. The systematic block tableau is one of the simplest possible tableau formulations which is still complete. Due to its inefficiency it is only interesting for theoretical investigations. Our presentation is close to that of Smullyan [Smu68]; however, we use a (systematic) block tableau instead of a (systematic) analytic tableau. In a systematic block tableau, there is no nondeterminism during the proof search. This is achieved by (i) providing a mechanism to select the next formula to be expanded (ii) providing a mechanism to determine the closed term to be used for the quantifier rules. It can be shown that in the

Block Tableau Rules		Sequent Calculus Rules	
T-rule	F-rule	L-rule	R-rule
$\frac{S, T(A \wedge B)}{S, TA, TB}$	$\frac{S, F(A \wedge B)}{S, FA \mid S, FB}$	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge_L$	$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge_R$
$\frac{S, T(A \vee B)}{S, TA \mid S, TB}$	$\frac{S, F(A \vee B)}{S, FA, FB}$	$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee_L$	$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_R$
$\frac{S, T(A \Rightarrow B)}{S, FA \mid S, TB}$	$\frac{S, F(A \Rightarrow B)}{S, TA, FB}$	$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow_L$	$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow_R$
$\frac{S, (T\neg A)}{S, FA}$	$\frac{S, F(\neg A)}{S, TA}$	$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg_L$	$\frac{\Gamma, A \vdash \Delta}{\Gamma, \neg A, \Delta} \neg_R$
$\frac{S, T(\forall x.A)}{S, TA[t/x], T(\forall x.A)}$	$\frac{S, F(\forall x.A)}{S, FA[c/x], F(\forall x.A)}$	$\frac{\Gamma, A[t/x], \forall x.A \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta} \forall_L$	$\frac{\Gamma \vdash A[c/x], \Delta}{\Gamma \vdash \forall x.A, \Delta} \forall_R$
$\frac{S, T(\exists x.A)}{S, TA[c/x], T(\exists x.A)}$	$\frac{S, F(\exists x.A)}{S, FA[t/x], F(\exists x.A)}$	$\frac{\Gamma \vdash A[t/x], \exists x.A, \Delta}{\Gamma \vdash \exists x.A, \Delta} \exists_L$	$\frac{\Gamma, A[c/x] \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta} \exists_R$

Table 7.1: Correspondence block tableau and sequent calculus

case that the method produces an infinite branch, the original formula set was satisfiable, provided a *fair strategy* is chosen, that is, a strategy that does not prefer certain formulas of the branch.

This is due to Hintikka's Lemma, which states that every Hintikka set is satisfiable. Before stating Hintikka's Lemma, we need some additional definition.

Definition 7.2.2 (Herbrand Universe, Herbrand Interpretation). *Let S be a set of formulas of a language L. Let SC denote the set of constants occurring in S. The constant base of S is SC if SC is non-empty, and the singleton set $\{a\}$ if $SC = \emptyset$. The function base SF of S is the set of function symbols occurring in S with arities > 0. Then the* Herbrand universe *of S is the set of terms defined inductively as follows.*

1. *Every element of the constant base of S is in the Herbrand universe of S.*
2. *If $t_1, \ldots, t_n$ are in the Herbrand universe of S and f is an n-ary function symbol in the function base of S, then the term $f(t_1, \ldots, t_n)$ is in the Herbrand universe of S.*

A Herbrand interpretation *for S is an interpretation $\mathcal{I}$ with the following properties:*

1. *$\mathcal{I}$ maps every constant in SC to itself.*
2. *$\mathcal{I}$ maps every function symbol f in SF with arity $n > 0$ to the n-ary function that maps every n-tuple of terms $\langle t_1, \ldots, t_n \rangle \in U^n$ to the term $f(t_1, \ldots, t_n)$.*

Definition 7.2.3 (Downward saturated set). *Let S be a set of first order sentences and $\mathcal{U}$ the Herbrand universe of S. The set S is called* downward saturated *provided*

1. *if S contains an α, then it contains all its α-subformulas*
2. *if S contains a β, then it contains at least one of its β-formulas*
3. *if S contains a γ, then it contains all $\gamma(t)$ with $t \in \mathcal{U}$*
4. *if S contains a δ, then it contains a $\delta(c)$ with c being a constant in $\mathcal{U}$.*

Definition 7.2.4 (Hintikka set)**.** *A Hintikka set is a downward saturated set which does not contain an atomic formula and its negation.*

Lemma 7.2.5 (Hintikka's Lemma)**.** *Every Hintikka set is satisfiable.*

Proof. Let S be a Hintikka set. Let H be the set of ground literals in S. As H does not contain a literal L and its negation, it defines a Herbrand interpretation $\mathcal{H}$. We show by induction over the structure of the formula that $\mathcal{H}$ maps every element of S to the truth value $\top$ (true). If F is atomic and $F \in S$, then $\mathcal{H}(F) = \top$ by construction. Suppose now that F is non-atomic.

1. F is of type α. By definition 7.2.3, every $\alpha_i \in S$. By induction hypothesis, $\mathcal{H}(\alpha_i) = \top$. Thus, by Lemma 4.1.31 $\mathcal{H}(F) = \top$.
2. F is of type β. By definition 7.2.3, some $\beta_i \in S$. By induction hypothesis, $\mathcal{H}(\beta_i) = \top$. Thus, by Lemma 4.1.31 $\mathcal{H}(F) = \top$.
3. F is of type γ. By definition 7.2.3, $\mathcal{H}(t) = \top$ for any $t \in \mathcal{U}$. Since $\mathcal{U}$ is the universe of $\mathcal{H}$ and because $\mathcal{H}$ is a Herbrand interpretation, for all variable assignments $\mathcal{A}$ $\mathcal{H}^{\mathcal{A}}(F') = \mathcal{H}(F'[x/\mathcal{A}(x)]) = \top$. Therefore, $\mathcal{H}(F) = \top$.
4. F is of type δ. By definition 7.2.3, $\mathcal{H}(F[x/c]) = \top$ for some constant $c \in \mathcal{T}$. Therefore, $\mathcal{H}(F) = \top$.

□

For the node selection we always select the leftmost leaf of minimal depth. The problem of instantiating quantifiers is solved by using a complete ordering on the set of closed terms and enumerating all possible instances in a systematic way, thereby not preferring any formula. This can be achieved by treating the set of unexpanded formulas in a priority queue, always selecting the first formula from the queue, inserting new copies at the end of the queue, and always producing the next instance with respect to the ordering. Thus, an unfair treatment as shown in Example 7.2.6 is no longer possible, and all possible terms are generated.

Example 7.2.6. *The following example shows the effect of an unfair treatment of formulas: Even though the formula is unsatisfiable, an infinite branch is generated.*

$$\begin{array}{c}
\vdots \\ \hline
P(a), P(b), \forall x.P(x), Q \wedge \neg Q \\ \hline
P(a), \forall x.P(x), Q \wedge \neg Q \\ \hline
\forall x.P(x), Q \wedge \neg Q
\end{array}$$

To generate all possible instances in a systematic way, we attach a number to each formula. Intuitively, this number keeps track of which instances have already been generated in order to always generate new instances. Taking these modifications into account allows the definition of the initial systematic block tableau for a given finite set S of formulas as well as all systematic block tableau we can obtain from it by expansion. To be able to express the systematic expansion strategy we use sequences instead of sets of formulas.

Definition 7.2.7 (Systematic Block Tableau)**.** *Let ξ be a mapping from $\mathbb{N}_0$ onto the set of ground terms and let S be a finite sequence of closed first order formulas. The* systematic block tableau *sequence of S w.r.t. ξ is the following sequence $\mathcal{T}$ of tableau for S:*

- *The one-node tableau T_0 with root S where each formula is labeled with the number* 0 *is an element of $\mathcal{T}$.*
- *If T_n is the n-th element in $\mathcal{T}$, let N be the leftmost leaf node of T_n which is not atomically closed, i.e. does not contain complementary literals. N has the form $l_1, \dots, l_n, G_1, \dots, G_m$ where l_i are literals and all G_i are non-literals (i.e. G_1 is the leftmost formula which is not a literal). Now expand N as follows:*
  1. *if G_1 is of type α, extend N by the node*
  $$l_1, \dots, l_n, \alpha_1, \alpha_2, G_2, \dots, G_m \tag{7.1}$$
  where α_i have label 0*.*
  2. *if G_1 is of type β, extend N by the following two branches:*
  $$l_1, \dots, l_n, \beta_1, G_2, \dots, G_m \mid l_1, \dots, l_n, \beta_2, G_2, \dots, G_m \tag{7.2}$$
  where β_i have label 0 *and* $\mid$ *indicates branching.*
  3. *if G_1 is of type γ, then extend N by the node*
  $$l_1, \dots, l_n, \gamma(\xi(k)), G_2, \dots, G_m, G_1 \tag{7.3}$$
  where k is the label of G_1, $\gamma(\xi(k))$ has label 0 *and the new copy of G_1 has label $k+1$.*
  4. *if G_1 is of type δ, then extend N by the node*
  $$l_1, \dots, l_n, \delta(c), G_2, \dots, G_m \tag{7.4}$$
  where c is the smallest constant modulo ξ not occurring in T.

Notation 7.2.8. *Let N be a node of a systematic block tableau. We call the label of N the block of the node N.*

Figure 7.2 shows an example of a systematic block tableau with $\xi(0) = a$, $\xi(1) = b$ and $\xi(2) = c$. While being inefficient, all instances needed to close the branches are eventually constructed. The systematic block tableau has the following property:

Lemma 7.2.9 (Fairness of Systematic Block Tableau)**.** *Let B be an open branch of a saturated systematic block tableau, N be a node of B and F be a non-atomic formula of the block of N. Then there exists a node N' below N at which F is selected for expansion, i.e., is the first non-atomic formula of the block corresponding to N. In particular, if F is of type γ, then F occurs infinitely many often as first formula on the branch B.*

Proof. Let N be of the form

$$L_1, \dots, L_m, G_1, \dots, G_m, F, G_{m+1}, \dots, G_k \tag{7.5}$$

where G_i are non-atomic. The proof is by induction on the sum k of the rank of the $L_1, \dots, L_m, G_1, \dots, G_m$.

Base Case: If $k = 0$, then F is a literal and nothing is to show.

$$\vdots$$

$$\dfrac{\dfrac{P(c,a), \neg P(a,b), P(c,b), \neg P(b,a) \vee \neg P(b,b), \forall y.P(c,y), \forall z.\neg P(z,a) \vee \neg P(z,b)}{P(c,a), \neg P(a,b), P(c,b), \forall z.\neg P(z,a) \vee \neg P(z,b), \forall y.P(c,y)}}{P(c,a), \neg P(a,b), \forall y.P(c,y), \forall z.\neg P(z,a) \vee \neg P(z,b)}$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{P(c,a), \neg P(a,a), P(c,a), \forall z.\neg P(z,a) \vee \neg P(z,b), \forall y.P(c,y)}{P(c,a), \neg P(a,a), \forall y.P(c,y), \forall z.\neg P(z,a) \vee \neg P(z,b)} \qquad \vdots}{P(c,a), \neg P(a,a) \vee \neg P(a,b), \forall y.P(c,y), \forall z.\neg P(z,a) \vee \neg P(z,b)}}{P(c,a), \forall z.\neg P(z,a) \vee \neg P(z,b), \forall y.P(c,y)}$$

$$\dfrac{\dfrac{\forall y.P(c,y), \forall z.\neg P(z,a) \vee \neg P(z,b)}{\exists x.\forall y.P(x,y), \forall z.\neg P(z,a) \vee \neg P(z,b)}}{(\exists x.\forall y.P(x,y)) \wedge (\forall z.\neg P(z,a) \vee \neg P(z,b))}$$

Figure 7.2: Systematic Block Tableau (written upside down)

Step Case: If $k = n$, G_1 is expanded. We perform a case distinction on G_1.

1. G_1 is of type α: According to Definition 7.2.7 the successor node of N is of the form (7.1), where $\sum_{i=1}^{m} rk(G_i) = k - 1$, for which the induction hypothesis is applicable.

2. G_1 is of type β: According to Definition 7.2.7 the successor nodes of N are of the form (7.2), and in both branches $\sum_{i=1}^{m} rk(G_i) = k - 1$ and one of them is open. Thus the assertion holds by induction hypothesis.

3. G_1 is of type γ: According to Definition 7.2.7 the successor node of N is of the form (7.3) and we have $\sum_{i=1}^{m} rk(G_i) = k - 1$. Thus the induction hypothesis is applicable.

4. G_1 is of type δ: According to Definition 7.2.7 the successor node of N is of the (7.4) and we have $\sum_{i=1}^{m} rk(G_i) = k - 1$. Thus the induction hypothesis is applicable.

□

Definition 7.2.10 (Saturated systematic block tableau)**.** *Let $\mathcal{T}$ be a systematic block tableau sequence for a set of first order formulas S. With T^* we denote the smallest tree containing all tableau $\mathcal{T}$ as initial segments; T^* is called a* saturated systematic block tableau *for S.*

Lemma 7.2.11. *For any open branch B of a saturated systematic block tableau for S, the set of formulas on B is a Hintikka set.*

Proof. By case distinction over the structure of F. Let F be a formula on an open branch B.

Base Case: If F is atomic, $\neg F$ is not on B, as B would otherwise be closed.

Step Case:

1. F is of type α: There exists a node N on B at which F is the first non-atomic formula of the block of N. Then by construction, both α_1 and α_2 occur at B.

2. F is of type β: There exists a node N on B at which F is the first non-atomic formula of the block of N. Then by construction, β_j occurs on B for some $j \in 1, 2$.

3. F is of type γ. Let $t \in U$. By construction, F occurs infinitely many often as first non-atomic formula in a block on B. Moreover, by construction of ξ there is a $k \in \mathbb{N}_0$ such that $\xi(k) = t$. Therefore, $\gamma(t)$ on B.

4. F is of type δ: By construction, there exists exactly one $t \in U$ such that $\delta(t)$ is on B.

□

Lemma 7.2.12 (Systematic Block Tableau Completeness)**.** *If S is an unsatisfiable set of first-order sentences, then there exists a finite atomically closed systematic block tableau for S.*

Proof. Let T be a saturated systematic tableau for S. Assume that T contains an atomically open branch B. In this case, there exists a Hintikka set for the set of formulas S' on B (Lemma 7.2.11), and therefore a model for S' (Lemma 7.2.5). As $S \subset S'$, this would also be a model for S, contradicting the assumption. Hence, every branch of T must be atomically closed.

As each closed branch is finite and the branching of any tableau is finite, T must be finite. □

Remark 7.2.13. *The completeness proof above reveals the following properties of systematic assertion level tableau:*

- *The completeness proof also works when allowing for non-atomic closures, i.e., to close a branch when two formulas F and $\neg F$ occur on a branch.*

- *It is sufficient that a single branch of the tableau forms a Hintikka set. Therefore, selecting always the leftmost non-closed leaf for expansion does not destroy the completeness proof. This way, one would obtain a proof procedure similar to the prover* Lean$T^{A}P$[1]

7.2.3 Systematic Assertion Level Tableau

In this section, we define a special calculus which is close to the assertion level and show its completeness. The main difference to the systematic block tableau defined above is that several decomposition steps are performed in a row, but only a copy of the top-level formula is kept. This will give us exactly our assertion level derivations, where each such sequence can be replaced by a derived rule. Due to the fact that formulas have been skolemized beforehand, there are no issues concerning the δ-rule. However, let us point out that if the skolemization is performed at runtime, a liberalized version of the δ-rule (see [BHS93]) is needed, as the following example illustrates.

[1]without the free variables

Example 7.2.14. *The following example illustrates the importance of the liberalization of the δ-rule. Because we do not keep intermediate formulas, the following proof cannot be performed at the assertion level without the liberalization of the δ-rule*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(c,a),\forall y.P(c,y)\vdash P(c,a)}}{\forall y.P(c,y)\vdash P(c,a)} \qquad \dfrac{\dfrac{}{P(c,b),\forall y.P(c,y)\vdash P(c,b)}}{\forall y.P(c,y)\vdash P(c,b)}}{\forall y.P(c,y)\vdash P(c,a)\wedge P(c,b)}}{\forall y.P(c,y)\vdash \exists z.P(z,a)\wedge P(z,b)}}{\exists x.\forall y.P(x,y)\vdash \exists z.P(z,a)\wedge P(z,b)}$$

Removing the intermediate formula $\forall y.P(c,y)$ requires the expansion of the formula $\exists x.\forall y.P(x,y)$ twice. However, in order to close both branches the same parameter for x is needed. The liberalized δ-rule allows for the following proof[2]*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(c,a)\vdash P(?z,a)}}{\forall y.P(c,y)\vdash P(?z,a)}}{\exists x.\forall y.P(x,y)\vdash P(?z,a)} \qquad \dfrac{\dfrac{\dfrac{}{P(c,b)\vdash P(?z,b)}}{\forall y.P(c,y)\vdash P(?z,b)}}{\exists x.\forall y.P(x,y)\vdash P(?z,b)}}{\exists x.\forall y.P(x,y)\vdash P(?z,a)\wedge P(?z,b)}}{\exists x.\forall y.P(x,y)\vdash \exists z.P(z,a)\wedge P(z,b)}$$

In particular, the liberalized δ-rule ensures that a single parameter is used for the same formula.

Even with skolemized formulas, there is still the problem of producing all possible instances for the γ-rules of a branch in a systematic way. Suppose first the case that all quantifiers are at top-level. In this case, a suitable enumeration of all possible terms can be achieved by the use of the Cantor pairing function[3].

Definition 7.2.15 (Cantor Pairing Function)**.**

$$\pi^{(k)} : \mathbb{N}_0^k \to \mathbb{N}_0, \pi^{(1)}(x) = x, \pi^{(n)}(x) = \pi(\pi^{(n-1)}(k_1, \ldots, k_{n-1}), k_n) \tag{7.6}$$

where

$$\pi(k_1, k_2) = \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2 \tag{7.7}$$

The Cantor pairing function visits all points of the $\mathbb{N}^k$ in a systematic way, as illustrated in Figure 7.3 for the case $k = 2$. The Cantor pairing function is invertible:

Theorem 7.2.16 (Invertibility)**.** *The Cantor pairing function is invertible. Its inverse is given by the function*

$$\left(\pi^{(k)}\right)^{-1} : \mathbb{N} \to \mathbb{N}^k \tag{7.8}$$

The ith component of the inverse is denoted by

$$\pi_i^{(k)} : \mathbb{N} \to \mathbb{N} \tag{7.9}$$

[2]Note that instead of using the meta-variable, we can also directly use c as the skolem constant is local to the formula

[3]as in the proof of reducing the denumerability of $\mathbb{N}_0^2$ to the denumerability of $\mathbb{N}_0$

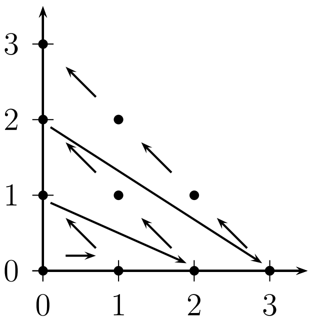


Figure 7.3: Illustration of the Cantor pairing function for $k = 2$

which is defined by

$$\pi_i^{(k)} = pr_i^{(k)} \circ \left(\pi^{(k)}\right)^{-1} \tag{7.10}$$

where is the ith projection: $pr_i^{(k)}(x_1, \ldots, x_k) = x_i$.

For the systematic assertion level tableau, each formula on the tableau is labeled with an integer. In addition, there are distinguished formulas, called *focused formula*, denoted by $\underline{F}$, which are never duplicated and need to be fully processed before another formula on the branch can be expanded. The only purpose of the focus is to avoid to copy the formula in the case of γ formulas. The more complex enumeration of terms is incorporated by labeling focused formulas with a triple (i, j, k). Intuitively, i corresponds to the number instances that have already been produced for a given assertion, $j-1$ to the quantifiers that have already been processed while the formula was in the focus, and k the overall number of the γ-quantifiers of the assertion which keeps unchanged during the processing. The purpose of the last two components is to select the corresponding enumeration function and projection. We illustrate the enumeration process with an example:

Example 7.2.17. *Figure 7.3 illustrates the systematic enumeration of all pairs of integers graphically, as generated by the Cantor pairing function.*

k_1	k_2	$\pi(k_1, k_2)$	$\left(\pi^{(2)}\right)^{-1}(\pi(k_1, k_2))$
0	0	0	$(0, 0)$
1	0	1	$(1, 0)$
0	1	2	$(0, 1)$
2	0	3	$(1, 1)$

$\left(\pi^{(2)}\right)^{-1}$*:* $(0, 0), (1, 0), (0, 1), (2, 0)$. *For a formula* $\forall x.\forall y.P(x, y)$ *this results in the following derivation under the ordering* $\xi(0) = a, \xi(1) = b$*:*

$$\begin{array}{cl}
\forall x.\forall y.P(x,y) & \\
| & \\
\underline{\forall x.\forall y.P(x,y)}, \forall x.\forall y.P(x,y) & \textit{(0,1,2)} \\
| & \\
\underline{\forall y.P(a,y)}, \forall x.\forall y.P(x,y) & \textit{(0,2,2)} \\
| & \\
\underline{P(a,a)}, \forall x.\forall y.P(x,y) & \textit{(0,3,2)} \\
| & \\
P(a,a), \underline{\forall x.\forall y.P(x,y)}, \forall x.\forall y.P(x,y) & \textit{(1,1,2)} \\
| & \\
P(a,a), \underline{\forall x.\forall y.P(x,y)}, \forall x.\forall y.P(x,y) & \textit{(1,2,2)} \\
| & \\
P(b,a), \underline{\forall y.P(a,y)}, \forall x.\forall y.P(x,y) & \textit{(1,3,2)}
\end{array}$$

Definition 7.2.18 (Systematic Assertion Level Tableau)**.** *Let ξ be a mapping from $\mathbb{N}_0$ onto the set of ground terms and let S be a finite sequence of closed first order formulas.*

The systematic assertion level tableau *sequence of S w.r.t. π is the following sequence $\mathcal{T}$ of tableau for S:*

- *The one-node tableau T_0 with root S where each formula is labeled with the number 0 is an element of $\mathcal{T}$.*
- *If T_n is the n-th element in $\mathcal{T}$, let N be the leftmost leaf of T_n which is not atomically closed, i.e. does not contain complementary literals.*
  - *If no formula attached to N is focused, i.e.,*

    $$N = l_1, \ldots, l_n, G_1, \ldots, G_m \tag{7.11}$$

    focus the leftmost non-literal by expanding the node N to

    $$l_1, \ldots, l_n, \underline{G_1}, G_2, \ldots, G_n, G_1 \tag{7.12}$$

    where $\underline{G_1}$ gets the label $(i, 1, k)$ if G_1 has label i, where k denotes the number of γ-quantifiers in G_1. In addition, add a copy of the formula G_1 at the end of the sequence and assign the label $i + 1$ to it. If all formulas are literals, then stop.
  - *Otherwise the block attached to N is of the form*

    $$l_1, \ldots, l_n, \underline{G_1}, G_2, \ldots, G_m \tag{7.13}$$

    where l_i are literals (i.e. G_1 is the leftmost formula which is not a literal). Now expand N as follows:

    1. *if G_1 is of type α with label (i, j, k), extend N by the node*

       $$l_1, \ldots, l_n, \underline{\alpha_1}, \underline{\alpha_2}, G_2, \ldots, G_n \tag{7.14}$$

       where both α_i have label (i, j, k).
    2. *if G_1 is of type β with label (i, j, k), extend N by the following two branches:*

       $$l_1, \ldots, l_n, \underline{\beta_1}, G_2, \ldots, G_m \mid l_1, \ldots, l_n, \underline{\beta_2}, G_2, \ldots, G_m \tag{7.15}$$

       where β_i have label (i, j, k).
    3. *if G_1 is of type γ, then extend N by the node*

       $$l_1, \ldots, l_n, \underline{\gamma(\xi(\pi_i^{(j)}(k)))}, G_2, \ldots, G_m \tag{7.16}$$

       where (i, j, k) is the label of G_1, $\gamma(\xi(\pi_i^{(j)}(k)))$ has label $(i, j + 1, k)$.
    4. *if G_1 is of type δ, then extend N by the node*

       $$l_1, \ldots, l_n, \delta(c), G_2, \ldots, G_m \tag{7.17}$$

       where c is the skolem constant corresponding to the formula δ.

    Remove all foci on literal formulas.

Figure 7.4 summarizes the rules graphically, where the context Γ is used to collect literal formulas. Moreover, we allow the instantiation with any ground term t in the γ-rule, but require that all terms are eventually generated.

$$\dfrac{\Gamma; \underline{\alpha}, S}{\underline{\alpha_1}, \underline{\alpha_2}, S} \qquad \dfrac{\Gamma; \underline{\beta}, S}{\Gamma; \underline{\beta_1}, S \mid \Gamma; \underline{\beta_2}, S} \qquad \dfrac{\Gamma; \underline{\gamma}, S}{\Gamma; \underline{\gamma(t)}}\ \text{FOR ANY GROUND TERM } t \qquad \dfrac{\Gamma; \underline{\delta}, S}{\Gamma; \underline{\delta(f_\delta)}, S}$$

$$\dfrac{\Gamma; \underline{s_1}, S}{\Gamma, s_1; S}\ s_1 \text{ ATOMIC}, \neg s_1 \notin \Gamma \qquad \dfrac{\Gamma; \underline{s_1}, S}{}\ s_1 \text{ ATOMIC}, \neg s_1 \in \Gamma \qquad \dfrac{\Gamma; s_1, S}{\Gamma; \underline{s_1}, S, s_1}$$

Figure 7.4: Assertion Level Tableau

Definition 7.2.19 (Saturated systematic assertion level tableau). *Let $\mathcal{T}$ be a systematic block tableau sequence for a set of first order formulas S. With T^* we denote the smallest tree containing all tableau $\mathcal{T}$ as initial segments; T^* is called a* saturated systematic assertion level tableau *for S.*

As before, we can show that each assertion is considered infinitely many often in an open branch:

Lemma 7.2.20 (Fairness Systematic Assertion Level Tableau). *Let B be an open branch of a saturated systematic assertion level tableau and. Let N be a node of B and F be a non-atomic formula of the block of N. Then the assertion corresponding to F occurs infinitely many often as first formula on the branch B.*

Proof. For the case that F is unfocused, the proof is similar to the proof of Lemma 7.2.9. The interesting case is that F is already focused. Then consider the first focus step above N, which focuses a formula F'. F is a subformula of F' and N contains a copy of F', and F' is copied by the focusing step. □

What remains to be shown to be able to reuse the completeness argument of the block tableau is to show that there is an open branch that is a Hintikka set. In the case of a systematic assertion level tableau, this is also the case if all quantifiers are at top-level, which can always be achieved by transformation rules and in which the completeness proof from the previous section can be reused. However, for general assertions, this needs not necessarily be the case, as the following example shows:

Example 7.2.21. *Consider the proof attempt of a formula δ using the assertion $\phi := (\forall x.Q(x)) \vee (\forall z.P(z))$, as shown below. Every branch is open. However, by selecting the left branch at the first branching, the right branch at the second branch, and then always the left branch for subsequent branchings we obtain an open branch which is not a Hintikka set, as it contains $\forall z.P(z)$ but not all instances of it.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{Q(a), \underline{\forall x.Q(x)}, \varphi \vdash \delta} \qquad \dfrac{\vdots}{Q(a), \underline{\forall z.P(z)}, \varphi \vdash \delta}}{Q(a), \underline{\forall x.Q(x) \vee \forall z.P(z)}, \varphi \vdash \delta}}{Q(a), \varphi \vdash \delta}}{\underline{\forall x.Q(x)}, \varphi \vdash \delta} \qquad \dfrac{\dfrac{\vdots}{P(a), \varphi \vdash \delta}}{\underline{\forall z.P(z)}, \varphi \vdash \delta}}{\dfrac{\underline{\forall x.Q(x) \vee \forall z.P(z)}, \varphi \vdash \delta}{\forall x.Q(x) \vee \forall z.P(z) \vdash \delta}}$$

However, this can be repaired under the following condition:

Definition 7.2.22 (Assertion)**.** *Let F be a formula. Then F is called* assertion *if it does not contain a subformula of type β that contains a γ quantifier.*

Remark 7.2.23. *Note that all formulas can always be transformed to an equivalent form in which they satisfy the assertion property.*

Lemma 7.2.24. *For any open branch B of a saturated systematic assertion level tableau for S, where each $s \in S$ is an assertion. Then the set of formulas on B is a Hintikka set.*

Proof. Let B be an open branch on which F occurs, say on node N. Without loss of generality we can assume that F is the first focused formula on N (Lemma 7.2.20).

1. F is of type α. If F is the first focused formula, then its successor N' is of the form (7.14), and both α_i occur on N'.

2. F is of type β. If F is the first focused formula, then its successor N' is of the form (7.14), and either β_1 or β_2 occurs on N'.

3. F is of type δ. If F is the first focused formula, then its successor N' is of the form (7.17) and $\delta(c)$ is on B for some c.

4. F is of type γ. Then the assertion F' corresponding to F occurs infinitely many often on B. Moreover, as no γ-formulas are β-related, they always appear on the same branch.

□

Theorem 7.2.25 (Completeness of Systematic Assertion Level Tableau)**.** *If S is an unsatisfiable set of assertions. Then there exists a finite atomically closed systematic assertion level tableau for S.*

Proof. As the proof of Theorem 7.2.12. □

By construction, a systematic assertion level tableau automatically constructs an assertion level derivation.

Theorem 7.2.26 (Assertion Level Derivation)**.** *Let $S_1, \ldots, S_n$ be assertions and G be a conjecture to be shown. Let Π be a systematic assertion level derivation for $\neg G, S_1, \ldots, S_n$. Then Π is an assertion level derivation.*

Proof. By induction over the length of the derivation. □

Moreover, it is now easy to replace each focused derivation by a rule application of the corresponding assertion. While the goal in the presentation above is also treated like an assertion, it is also possible to restrict the focusing to assumptions and to treat the goal separately and to switch between assertion applications and rule applications on the remaining formulas. As long as fairness is guaranteed, our considerations above remain valid.

Remark 7.2.27. *Each focused derivation can be seen as the application of a new inference rule to which it corresponds. Within our setting above, we required an assertion to be always decomposed completely. This way we obtain a maximal caching of such derivations in inferences. Note however that stopping the decomposition earlier does not destroy completeness, as long as the formulas are further decomposed later.*

We now want to go one step further and consider the typical case that a consistent theory of assertions is given, where consistency is defined as follows:

Definition 7.2.28 (Consistency)**.** *Let Γ be a set of formulas. Γ is called consistent if there is no formula φ such that $\Gamma \vdash \varphi$ and $\Gamma \vdash \neg\varphi$.*

The question we want to explore next is whether we can further restrict the application of assertions. The following theorem establishes a connection between a goal literal G and a consistent set of formulas Γ:

Theorem 7.2.29 (Existence of positive subformulas)**.** *Let Γ be a consistent set of formulas, let G be a literal and let $\Gamma \vdash G$ be provable. Then there exists a positive occurrence of G in Γ.*

Proof. Let Π_1 be a proof of $\Gamma \vdash G$. Every axiom in Π_1 is of the form $\Gamma, A \vdash A, \Delta$ for some proposition A. Suppose that Γ does not contain a positive occurrence of G. Let Π_2 be a partial derivation involving formulas of Γ followed by the application of the axiom rule. As G does not occur as positive occurrence of G, Π_2 will not generate the literal G on the left-hand side of the sequent. Thus G is not used within the axiom rule. We can therefore obtain a proof Π' in which all occurrences of G on the right-hand side have been replaced by $\neg G$. However, this contradicts the assumption that Γ is consistent. □

The property means always an inference rule where some conclusion is instantiated is applicable, i.e., that we can require that at least one inference node is instantiated.

7.3 Summary

In this chapter we gave a proof theoretic characterization of assertion level derivations by introducing the notion of a systematic assertion level tableau and showed soundness and completeness with respect to first order logic (Contribution A1(ii), Section 1.1). The relation between assertion level derivations and block tableaux has not been studied before and shows that for the first order fragment redundancy can be avoided by committing to assertion level operations. Moreover, it identifies the assertion level to be a rather calculus independent layer.

Part III

Proof Plans and Proof Strategies

8 Proof Plans

The organization of the proof search on top of a calculus is an essential task in the design of a powerful theorem proving system. To combine interaction and automation into a synergetic interplay and to bridge the gap between abstract level proof explanation and low-level proof verification, we provide an abstract proof data structure to maintain a proof attempt, including possible alternatives, called a *proof plan*. Alternatives can be both *horizontal* and *vertical*: Horizontal alternatives correspond to different reductions of a proof state, while vertical alternatives correspond to a subproof of a step at a different level of granularity. The explicit support of hierarchies, gaps and their refinement, as well as the information how to refine gaps, and the handling of meta-variables are the features that distinguish proof plans from other proof representations.

To communicate proof plans to the user, we exploit the relationship between proof plans and declarative proof scripts by the possibility to render a particular alternative of a proof plan as a declarative proof script. Similarly, we use underspecified declarative proof scripts as input representation for proof plans and show how they can be refined automatically.

Our proof language combines features of Mizar and Isar, with the difference that meta-variables are supported. This has the advantage that also proof plans that have been generated automatically can be translated into proof scripts. Moreover, we will extend the language later to support the specification of reasoning procedures and annotated inferences as a means to extend and adapt the power of the underlying system at runtime. This will result in a declarative proof language that fully supports the document-centric approach.

8.1 Textbook Proofs, Proof Plans, and Declarative Proofs

Before turning to the details of our approach, let us motivate the concepts by the correspondence between textbook proofs, proof plans, and declarative proof scripts. The following example is a textbook proof reproduced from [BS82]:

Theorem 8.1.1. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Proof. Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii)$x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$. Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$. In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. □

From a formal point of view, the proof consists of a sequence of statements that are hierarchically structured. For example, the main statement $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ is proved in two blocks, one for each set inclusion. Within a declarative proof language, these hierarchical structures are made explicit. Indeed, it is not difficult to translate this textbook proof to a (partial) declarative proof script, as shown on the left in Figure 8.1. Within the figure, the hierarchical structure is visualized by indentation. The partiality of the proof script results from the fact that information that is needed for a (formal) verification is still missing, even though the proof is already very detailed. This leads to the notion of a proof sketch (see [Wie04]) or a proof plan[1]. Here, we want to understand a *proof plan* as a data structure that supports the representation of proof outlines including gaps, as well as their mechanization, that is their *refinement*. Proof plans are directed, acyclic graphs, consisting of nodes representing the current context, and justifications, representing *proof commands*. Thus, declarative proof scripts can be seen as a particular rendering of a proof plan. In particular, it is easy to generate a proof plan from a declarative proof script. Figure 8.1 shows on the right the proof plan corresponding to the example proof script. In the proof plan, each hierarchy corresponds to a tree in a forest, which are linked together via forest edges (indicated by the dotted lines). Note that it would also be possible to represent a proof script as a single tree; however, we found the separate management of independent blocks in independent trees convenient. Even though proof plans and proof scripts are quite similar, let us stress the main advantages of proof plans over proof scripts:

- In a proof plan, the available context is made explicit.
- In a proof plan, each proof state is enriched with additional contextual information, such as labels for terms, or annotations such as needed for Rippling [BBHI03].
- The verification of reductions can be postponed, resulting in proof gaps. This allows the integration of external systems, such as computer algebra systems.
- Proof plans can represent multiple proof attempts in parallel, as well as on different levels of granularity. Thus, it is possible to represent the initial proof plan as well as its expanded form which can all be verified within one data structure.
- Proof plans support asynchronous proof checking.

In this sense, a proof plan is used for processing of declarative proof scripts, while a declarative proof script is used as input and output representation. Let us point out that by exploiting the relationship between declarative proof scripts and proof plans, it

[1]Note that within the interactive setting, the proof plan is not generated automatically by a proof planner, but manually written by a human. Historically, proof plans were developed before declarative proof scripts.

theorem $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

proof
  assume $x \in A \cap (B \cup C)$
    $x \in A \wedge (x \in B \vee x \in C)$
    $(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$
    $(x \in A \cap B) \vee (x \in A \cap C)$
  thus $x \in (A \cap B) \cup (A \cap C)$
  $A \cap (B \cup C) \subset (A \cap B) \cup (A \cap C)$
  assume $y \in (A \cap B) \cup (A \cap C)$.
    $y \in A \cap B \vee y \in A \cap C$.
    $y \in A \wedge (y \in B \vee y \in C)$
    $y \in A \wedge y \in B \cup C$
  thus $y \in A \cap (B \cup C)$
  $(A \cap B) \cup (A \cap C) \subset A \cap (B \cup C)$
qed by `Definition 1.1.1`

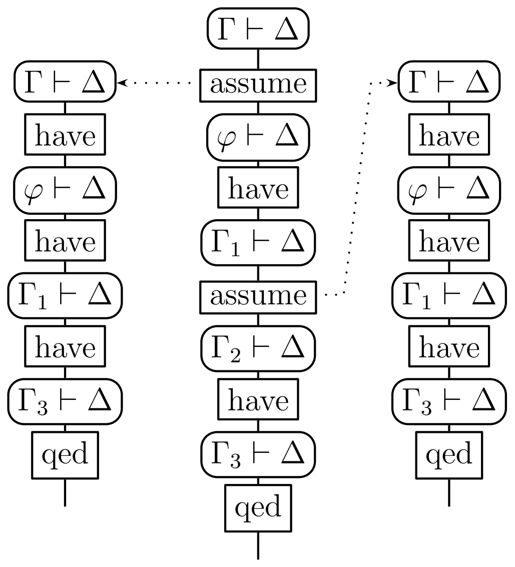


Figure 8.1: Partial declarative proof script obtained from the textbook proof

will become possible to synthesize declarative proof scripts from proof plans that have been constructed automatically. This is in particular useful in combination with our abstract assertion level reasoning, as the individual proof steps will be of an appropriate granularity, keeping the proof scripts readable.

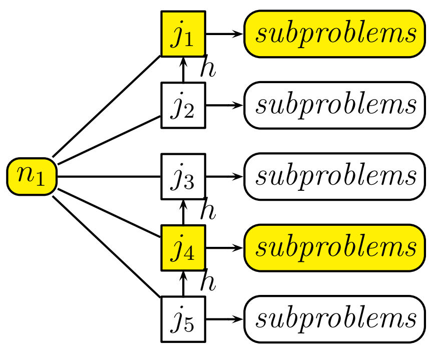
8.2 Proof Plans

As basic representation of proof plans we use the proof data structure $\mathcal{PDS}$ (see [ABD$^+$06, Die06] for details). The basic ideas are:

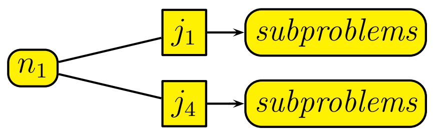
- Each conjectured *lemma* gets its own *proof tree*.
- In this *proof forest*, each lemma can be applied in each proof tree; either as a lemma in the narrower sense, or as an induction hypothesis in a possibly mutual induction process, see [Wir04].
- Inside its own tree, the lemma is a goal to be proved reductively. A *reduction* step reduces a *goal* to a conjunction of *sub-goals* w.r.t. a *justification*.
- Several reduction steps applied to the same goal result in alternative proof attempts, which either represent different proof *ideas* or the same proof idea with different *granularity* (or detailedness).

Each $\mathcal{PDS}$-tree is essentially a directed acyclic graph (dag) whose nodes are labeled with tasks. It has two sorts of links: *justification hyper-links* that represent some relation of a task node to its sub-task nodes, and *hierarchical edges* that point from justifications to other justifications which they refine. This definition allows for alternative justifications and alternative hierarchical edges.

Outgoing justifications of a node n, which are not connected by hierarchical edges, are OR-alternatives and represent alternative ways to tackle the same problem contained in n. They describe the horizontal structure of a proof. Horizontal OR-alternatives are motivated by the fact that we want to support proof search algorithms that explore several



(a) $\mathcal{PDS}$-node with all outgoing partially hierarchically ordered justifications, and j_1, j_4 in the set of alternatives. Justifications are depicted as boxes.



(b) $\mathcal{PDS}$-node in the $\mathcal{PDS}$-view obtained for the selected set of alternatives j_1, j_4.

Figure 8.2: An example $\mathcal{PDS}$ and one of its $\mathcal{PDS}$-views

alternatives in parallel, such as a resource bounded BFS-search, and to return all solutions that can be found within a given depth.

Hierarchical edges are used to construct the vertical structure of a proof. A proof may be first conceived at a high level of abstraction and then *expanded* to a finer level of granularity. Vice versa, *abstraction* means the process of successively contracting fine-grained proof steps to more abstract proof steps. For instance, in Figure 8.2(a), the edge from j_2 to j_1 indicates that j_2 refines j_1. The hierarchical edges distinguish between upper layer proof steps and their refinements at a more granular layer.

The structuring mechanism described above provides very rich and powerful means to represent and maintain the proof attempts during the search for the final proof. In particular, it supports two ways of OR alternatives, both alternative refinements at the vertical level (in the case that there are several possibilities to unpack an abstract proof idea such as finding a distinctive property to show that two structures are not isomorphic), as well as alternative refinements at the horizontal level (for example, when different theorems can be applied to the same task). In fact, such multidimensional proof attempts may easily become too complex for a human user, but since the user does not have to work simultaneously on different granularities of a proof, elaborate functionalities to access only selected parts of a $\mathcal{PDS}$ are helpful. They are required, for instance, for the user-oriented presentation of a $\mathcal{PDS}$, in which the user should be able to focus only on those parts of the $\mathcal{PDS}$ he is currently working on. At any time, the user can choose to see more details of some proof step or, on the contrary, he may want to see a coarse structure when he is lost in details and cannot see the wood for the trees.

One such functionality is a *$\mathcal{PDS}$-view* that extracts from a given $\mathcal{PDS}$ only the horizontal structure of the represented proof attempts, but with all its OR-alternatives. As an example consider the $\mathcal{PDS}$ fragments in Figure 8.2. Based on a $\mathcal{PDS}$-view it is then easy to iterate over all alternatives.

The node n_1 in the fragment on the left has two alternative proof attempts with different granularities. The fragment on the right gives a $\mathcal{PDS}$-view which results from selection of a certain granularity for each alternative proof attempt, respectively. The set of alternatives may be selected by the user to define the granularity on which he currently wants to inspect the proof. The resulting $\mathcal{PDS}$-view is a slice plane through the hierarchical $\mathcal{PDS}$ and is – from a technical point of view – also a $\mathcal{PDS}$, but without hierarchies, that is without hierarchical edges.

8.2.1 Handling Meta-Variables

Within the setting of automated proof search, meta-variables have been introduced to postpone choices of witness terms. This is also the case within our setting, in which γ-nodes introduce free variables in the free variable indexed task tree whose instantiation can be postponed.

As free variables are rigid, we need several different instances of a universally quantified variable to close a branch. Essentially, there are three main questions: (i) how many instances are needed to be able to close all branches, (ii) when to instantiate a variable, and (iii) how to propagate instantiations. Note that propagating a substitution that closes one branch might destroy the possibility to close another branch, and hence needs to be undone later. The following example illustrates the problem:

$$\dfrac{\dfrac{\dfrac{\vdash X = c, X = d}{\vdash X = c \lor X = d} \qquad X = d}{X = c \lor X = d \land X = d}}{\vdash \exists x.(x = c \lor x = d) \land x = d}$$

Closing first the left branch and propagating the substitution $X \mapsto c$ destroys the possibility to find a solution, unless backtracking is performed. To illustrate the need of several instances to close a proof attempt, consider the following example:

$$\dfrac{\dfrac{p(X) \vdash p(a) \qquad p(X) \vdash p(b)}{\vdash p(X) \Rightarrow p(a) \land p(b)}}{\vdash \forall x.(p(x) \Rightarrow p(a) \land p(b))}$$

Even though there exist some approaches which do not use free variables, such as [OS88, Smu68], the use of free variables in the automation of sequent calculi can be seen to be standard nowadays. Note that while the approaches without free variables require the correct guessing of the instances, they have the advantage that they do not require backtracking.

When using free variables, there are two possibilities how to work with free variables, which can be described as “unify as you go” and “unify at the end”.

Unify as you go (direct closure): In this kind of construction, whenever a closure is made that requires a binding of one or more free variables, the substitution is applied to all occurrences in the proof tree of those newly bound variables. This guarantees consistency of the bindings. Only one binding may be made to any free variable. Of course, choosing a wrong instantiation requires backtracking.

Unify at the end (delayed closure): In this construction, it is noted when a branch can close and what the corresponding binding is, but no propagation takes place. When every branch has such a potential closure the possible substitutions are combined. If they do not unify then alternative closures in one or more branches are sought. The disadvantage of this approach is that the prover might spend time on expanding a branch before detecting an inconsistency.

Direct Closure Approaches

A lean search procedure for tableau with free variables has been described in [BP95], which is still the basis of the tableau prover being used in ISABELLE (see [Pau99] for details). The prover uses iterative deepening on the number of free variables occurring in a branch and backtracking to obtain completeness. Substitutions are propagated among all branches of the tableau.

Delayed Closure Approaches

Several approaches have been proposed to overcome the need for backtracking in direct closure approaches. In the disconnection calculus [Bil96], copies of the literals with new variables are generated and the substitution is applied on the copies. A similar approach is taken in [BEF00]. However, both approaches work only for clausal form. A variant which works also for non-clausal forms is given by Beckert [Bec03]. The idea is to apply a reconstruction step after propagating a substitution, which reintroduces literals that were destroyed by the substitution. However, the techniques are rather complicated and it is not known whether an implementation has been attempted.

Fitting introduces the delayed closure approach and presents a basic implementation of it (see [Fit96]). At each step, the proof procedure tries to find a substitution closing all open branches simultaneously. Similarly, the evidence algorithm [DLM99] tries to close all branches simultaneously after each step. The disadvantage of this approach is that work to test for a closure might be duplicated.

THEOREMA uses a method called *solution lifting* [KJ01] to handle meta-variables. The general idea is to represent an ongoing proof attempt (including meta-variables) as an AND-OR tree, called *pre-proof*, and to attach substitutions locally to each node of the pre-proof. The local substitutions are lifted bottom up in the pre-proof by combining compatible substitutions, hoping to obtain an admissible substitution for the root node of the pre-proof. In this case, the proof is completed. If two substitutions are not compatible, the proof tree must be expanded somewhere under this node to obtain a new substitution. Such an expansion is always possible, as applying a substitution automatically introduces a new alternative in which the variables are uninstantiated. To avoid the use of the same substitution on the new branch, a restriction in the form of a disunification constraint is attached to the node that forbids previously used substitutions. This technique avoids backtracking and deletion of partially constructed solutions.

A similar approach, called *incremental closure*, has been described by Giese [Gie02], in which the closure is also computed in an incremental fashion. To that end, possible closures are locally computed for each node and represented as *unification constraints* and propagated upwards in the proof tree (taking some optimizations into account). Unification constraints are a universal technique for avoiding the global application of substitutions by decorating formulas with unification constraints. A unification constraint C is a conjunction of syntactic equalities between terms or formulae, written as

$$s_1 \equiv t_1 \wedge \ldots \wedge s_k \equiv t_k \tag{8.1}$$

where the symbol $\equiv$ is used for syntactic equality in the constraint language to avoid confusion with the standard equality $=$. The same technique is used to lift the ground simplification rules to the level of free variables. However, this time the constraints are attached locally to the formula, resulting in a constrained formula $\Phi \ll C$, meaning that Φ can be derived under the condition C. The constraints are taken into account when testing for closure, which is only possible if the combined constraint is satisfiable. For example, the goal $pX \ll X \equiv Z, \neg pa \ll Z \equiv B$ cannot be closed, as the first constraint requires $\sigma(Z) = \sigma(X) = a$, while the second constraint requires $\sigma(Z) = b$. To avoid that already simplified formulas are simplified again, disunification constraints are used. For simplification rules, a similar mechanism has been developed by Peltier, which attaches constraints in the form of syntactic equalities [Pel97, Pel99].

Extensions: In [Rüm08], Rümmer presents an extension of the incremental closure approach which does not only handle unification constraints, but arbitrary equality constraints in the fragment of Peano arithmetic (PA). The calculus is based on *constrained sequents* $\Gamma \vdash \Delta \Downarrow C$ which are sequents attached with a constraint C (in PA). Constraints are either unification constraints or introduced by the following rule

$$\frac{*}{\Gamma, \Phi_1, \ldots, \Phi_n \vdash \Psi_1 \ldots \Psi_m, \Delta \Downarrow \neg\Phi_1 \vee \ldots \vee \neg\Phi_n \vee \Psi_1 \ldots \vee \Psi_m} \textsc{Close} \tag{8.2}$$

in which Φ_i, Ψ_i do not contain uninterpreted predicates. As a consequence, the constraint resulting from a proof is always a formula in Presburger arithmetic. The idea is that closing the branch introduces the constraints C as indicated by the rule. A constrained sequent is identified with the formula $\bigwedge \Gamma \wedge C \Rightarrow \bigvee \Delta$. For the real numbers, a similar approach is taken by Platzer [Pla08], who embeds algebraic constraints over the reals in tableau calculi and uses an external quantifier elimination procedure to solve them.

General Discussion of our Choice

The representation of an ongoing proof attempt is essential for both the interactive and the automated proof search. However, the two worlds are usually conflictive: Efficient representations for the automated setting are often not suitable for the interactive setting. Similarly, while postponing choice points such as the introduction of constrained formulas might be valuable in a fully automatic setting, it may also lead to proof states which are difficult to understand. Therefore, constrained formulas are not an option within our setting.

The main advantage of propagating substitution is that this might enable further simplification steps which reduce both the length and number of branches in the proof tree (see e.g. [Mas98] for a discussion), while introducing backtrack points, which is a non-local operation. In our implementation, we apply a global substitution to the complete proof tree and propagate it along its refinements. This is the usual operation a user expects within an interactive setting; moreover, it is directly supported by the underlying calculus.

The $\mathcal{PDS}$ supports the operation of removing a proof step, including instantiation steps. As instantiations are propagated along all branches, proof steps are generally global operations. This poses the question of which other steps (on other branches) need to be deleted when deleting a single proof step. Instead of using a chronological notion of dependency which would delete all steps that have been performed after the step that is to be deleted, we use a more sophisticated dependency notion that does not delete the steps that are not affected when propagating a variable binding. For example, if a meta-variable x is instantiated, proof steps on other branches that do not involve the meta-variable x are not deleted. In particular, the deletion of a step is local to the current branch if no binding is propagated at all.

Note that while propagating substitutions globally, it is still possible to implement a delayed closure approach based on our data structure by postponing the application of a substitution. However, more sophisticated approaches, such as solution lifting or variants, are currently not supported. This is because such a mechanism would complicate the notion of a $\mathcal{PDS}$-view and make the implementation, as well as the translation of a $\mathcal{PDS}$ to a declarative proof script, more complicated. In particular, the proof generated by the decision procedure to check whether the constraints can be solved is not directly presentable to the user, contradicting our focus on a human-readable proof presentation.

8.3 A Declarative Proof Language

Declarative proof languages, such as Mizar or Isar, have been developed as a means to present and store a proof in a human-readable format. In this section, we develop a declarative proof language for ΩMEGA as an input and output presentation for proof plans. This has the advantage that gaps and their refinements are naturally supported, and that the checking itself is postponed and does therefore not stop at the position where a step cannot be justified. Before turning to our concrete language design, let us state the general requirements of a declarative proof language and discuss design choices.

- **Readability:** The proof script should contain enough explicit information to understand the proof without running the proof assistant.
- **Conciseness:** The proofs should be as short as possible to avoid unnecessary typing.
- **Continuous Checking** It is desirable to continue proof checking after an intermediate step could not be verified or parsed.
- **Maintainability:** The language should be implemented independently of the underlying prover.
- **Automation & Extendability** The language should allow for an efficient automation and provide facilities to extend the automation facilities of the prover.
- **Underspecification** The language should feature underspecified proof steps in the style of textbook proofs.
- **Simplicity** The language should be simple to learn.

Let us point out that even though existing declarative proof languages are quite similar, the language designer has several options, in particular when considering automation versus simplicity. On the one hand, it is desirable to have only very few language constructs to keep the language as simple as possible. This comprises both the number of proof commands, as well as the complexity of the justification hints, i.e., information describing how to close a proof step. Two extremes are the proof language Declare, which comes along with only three constructs (see [Sym99]) that support forward reasoning, or the system Sad, which provides no justification hints at all. On the other hand, this substantially complicates the verification process and reduces the efficiency of the proof checker. This is because in a forward proof, which is the style of most textbook proofs, the number of available assumptions grows with each step. Therefore, proof languages usually provide a mechanism to label formulas and to indicate the assumptions to be used to close a gap. It is also common to provide specific proof commands to refer to the previous fact without having to introduce a label, such as Isar's commands **moreover** and **ultimately** to collect facts to be used in the statement that follows **ultimately**. Of course, this makes the language more complex and more difficult to learn.

Similarly, there is the question whether one wants to support justification hints that go beyond the specification of assumptions, such as the specification of a tactic to be used to close a gap. Again, this makes the language more complex, but increases the freedom and allows the automation of gaps that cannot be automated otherwise.

In addition, there is the question when and how to process a specific proof command internally, which directly influences the structure of the input proof scripts. Consider for example a command **assume** to introduce a specific assumption. In MIZAR (see [TB85]), the command is directly executed on the current proof state and essentially corresponds to the natural deduction rule $\Rightarrow_I$. That is, it is expected that the goal is of the form $\Gamma \vdash A \Rightarrow B$, and the result of the transformation is a goal of the form $\Gamma, A \vdash B$. To be more general, the theory of *semantic correlates* is used to be able to go from $\Gamma \vdash A$ to $\Gamma, \neg A \vdash \bot$. Let us point out here that the command modifies the goal and introduces a new assumption and that for a successful processing the shape of the goal needs to have a particular form. Alternatively, it is possible to process an assumption lazily, i.e., to introduce a new lemma $\vdash A \Rightarrow B$ when a proof command **assume** A ... **thus** B is issued, and to check the connection to the current goal either afterwards or not at all. This has the advantage that only the conclusion needs to match the current goal; therefore, the resulting processing is more general and allows the user to write more general scripts. However, it has the disadvantage that the decomposition step is not performed automatically. A third possibility is to introduce a new fact $\neg A \vee A$ using the law of excluded middle, performing a case split afterwards and requiring to close the first case automatically.

Concerning the simplicity of the language, there is the question whether to support both the forward and backward style of proof, as it is possible to convert a backward proof into a forward proof and vice versa. Moreover, there is the choice whether to introduce abbreviating language constructs that avoid explicit labeling.

Finally, there are design choices concerning the concrete syntax of the proof language, which influence the complexity and efficiency of the underlying parser and error recovery from parsing errors. For example, ISABELLE/ISAR distinguishes between inner syntax for formulas and outer syntax for the theory structure and ISAR. This requires special markers for the inner syntax – formulas need to be written in quotes – but has the advantage that the syntax can be parsed independently of each other. Similarly, MIZAR requires the user to terminate a command using a semicolon. As before, the semicolon is an additional character to simplify the parsing and allows the further processing after an error.

8.3.1 Realization of the Language

Figure 8.1 showed an example of our proof language. In the context of the above discussion, our choices are:

- We explicitly support labels and justification hints in the form of **by** and **from**. The former refers to a strategy expression, e.g., the invocation of a tactic, and the latter allows the reference of labels that have been introduced before. Justification hints are optional, that is, can be left out by the user, and the order of the labels does not influence the checking process.

- We support both the forward and the backward style of proof. This will be important, as we will extend the proof language to specify proof strategies, which are

<document>	::= <theory>+
<theory>	::= **theory** <name> (**imports** <names>)? <theoryitem>+
<theoryitem>	::= <typedecl> | <definition> | <axiom> | <theorem> | <const>
<typedecl>	::= **newtype** <name>
<definition>	::= **definition** <name> <parserinfo>? <formula>
<const>	::= **const** <formula> <parserinfo>?
<axiom>	::= **theorem** <name> <formula>
<theorem>	::= **theorem** <name> <formula> <proof>?
<parserinfo>	::= [<pinfos>]
<pinfos>	::= <pinfo> | <pinfo> **,** <pinfos>
<pinfo>	::= **left** | **right** | **infixr** | **infixl** | **prec =** <number>

Figure 8.3: ΩMEGA theory language

sometimes expressed in a backward style more easily.

- To keep the language as simple as possible, we do not introduce further language constructs that avoid the explicit labeling of formulas, except special constructs to support reasoning chains involving binary operators, as needed for equality reasoning or inequality chains.
- Assumptions are lazily processed and no connection to the proof state is required. This provides maximal freedom in the structure of the proof scripts.
- We avoid special symbols to simplify parsing. Error recovery starts with the next keyword or after the next newline character of the proof script.
- To keep the language simple to type, we follow the idea in MIZAR and do not introduce a proof command for forward steps.
- To support asynchronous checking and error recovery, we transform the proof script to a proof plan, which makes the proof obligations explicit in the form of expansion tasks.

To show the extendability of our language, we will extend the language in Chapter 10 such that proof strategies can also be specified within the proof document.

Basic Theory Management

Proofs are constructed within the context of a mathematical theory, which consists of a signature of types and constants, and axioms. Consequently, we provide constructs to declare new types, definitions, and axioms. Each theory extends the base theory, which provides two base types for Booleans o and individuals ι, and a type constructor $\rightarrow$ to define function types. All other types are introduced by type definitions. Moreover, the base theory defines the usual logical connectives. Figure 8.3 shows the abstract syntax of ΩMEGA's theory language. Definitions introduce new constants based on the existing theory and dynamically extend the parsing facilities of the formula parser. Therefore, they can be attached with parsing information, such as the precedence of the introduced symbol, its associativity, and whether it is written infix, prefix, or postfix. An example of a simple theory is shown in Figure 8.4.

```
theory simplesets
const in::i->i->o[prec=200,infixl]
definition subsetdef[prec=210,infixl]
  !A,B. A subset B <=> (!x.  x in A => x in B)
definition setequaldef !A,B. A=B <=> (A subset B /\ B subset
A)
definition intersectiondef[prec=230,infixl]
  !A,B. x in A intersection B <=> x in A /\ x in B
definition uniondef[prec=220, infixl]
  !A,B,x.  x in A union B <=> x in A \/ x in B
```

Figure 8.4: Theory Simple Sets

Within a theory, theorems can be stated and proved using the proof language, whose abstract syntax is shown in Figure 8.5. We subsequently explain each of the constructs in detail and show how it modifies the current proof plan. Since we do not require justification hints at all and allow for arbitrarily large gaps, the verification of a single proof step can become time consuming. In the worst case a complete proof search has to be performed. To obtain adequate response times a given step is worked off in two phases: First, we perform a quick test, where we assume that the given step can be justified by the prover by a single inference application. This can be tested by a simple matching algorithm. If the test succeeds the step is sound, as the inference application is proved to be correct. Only if the quick test fails, a more comprehensive proof search is started. This mechanism tries to find the missing information needed to justify the step by performing a heuristically guided resource bounded search. If it is not able to find a derivation within the given bound, a failure is reported and further user interaction is necessary. Let us point out that our approach means that a given proof script can be checked efficiently provided that it contains enough information.

Justifications

We distinguish two kinds of justifications, *atomic justifications* and *nonatomic justifications*. Atomic justifications are the simplest kind of justification and consist of a

<proof>	::= **proof** <steps> **qed**
<steps>	::= (<ostep> <steps>)|<cstep>
<ostep>	::= <set>|<assume>|<have>|<cases>|<goal>|<consider>
<cstep>	::= (<goal>)$^+$ |ϵ
<by>	::= (**by** <name>)? | <proof>
<from>	::= (**from** <label> (, <label>)*)?
<sform>	::= <form> | **.** <binop> <form>
<assume>	::= **assume** <form> <steps> **thus** <form> <by> <from>
<have>	::= **have**? <sform> | <by> <from>
<cases>	::= (**case** <form> { <proof> })$^+$ <by> <from>
<goal>	::= **subgoal** <form> <from> <by>
<set>	::= **set** <var>=<form> (**,** <var>=<form>)*
<consider>	::= **consider** <form> <by> <from>

Figure 8.5: ΩMEGA proof script language

by annotation and/or a **from** justification. The **by** specifies an inference or tactic which shall be used to close the corresponding proof obligation. The **from** construct acts as filter on the current context Γ and specifies the assumptions to be used for the verification (i.e., all other assumptions are removed by weakening). Given a proof obligation δ and a context Γ containing derived facts and assumptions, a justification

$$\textbf{from } l_1, \ldots, l_n \textbf{ by } \ \mathit{exp} \tag{8.3}$$

generates the proof obligation

$$\begin{cases} l_1, \ldots, l_n \vdash \delta & \text{if } n > 0 \\ \Gamma \vdash \delta & \text{otherwise} \end{cases} \tag{8.4}$$

Complex justifications consist of an entire proof that is enclosed by the keywords **proof** and **qed** derives the specified fact. Subproofs introduce hierarchies in the proof plan and represent the specified fact at a lower granularity level. Note that there are two possibilities to introduce hierarchies in the proof plan: (i) by creating a new proof tree containing the more detailed proof and connecting it to the more abstract proof via a forest edge, and (ii) by inserting the more detailed proof in the same proof tree using a hierarchical edge. We have implemented the second approach.

Forward Steps

Forward steps are the steps that occur most frequently in declarative proof scripts. They derive a new formula φ, possibly labeled with label l, from the current proof context. To keep the proof scripts short, we do not assign a keyword to those steps and classify them internally as **have** steps. That is, given a proof state $\Gamma \vdash \Delta$, the proof command

$$a + 0 = 0 + a \tag{8.5}$$

results in the new proof situation $\Gamma, a + 0 = 0 + a \vdash \Delta$.

Quick Test: The quick test tries to justify the new fact by the application of the inference *name* to term positions in the formulas with labels $l_1, \ldots, l_n$ in the current task. As the result of an inference application is already known before the application of the inference, we can take advantage of this information to further restrict possible bindings. In the case that the quick test fails, the justification needs to be expanded, as shown by the dashed arrow below. Otherwise, the step is marked as correct.

$$\begin{tikzcd}
\boxed{\Gamma \vdash \Delta} \arrow[d, dash] & \boxed{\Gamma \vdash \varphi} \\
\boxed{\textbf{have } l : \varphi} \arrow[d] \arrow[ru, dashed] & \\
\boxed{\Gamma, l : \varphi \vdash \Delta} &
\end{tikzcd}$$

Assume

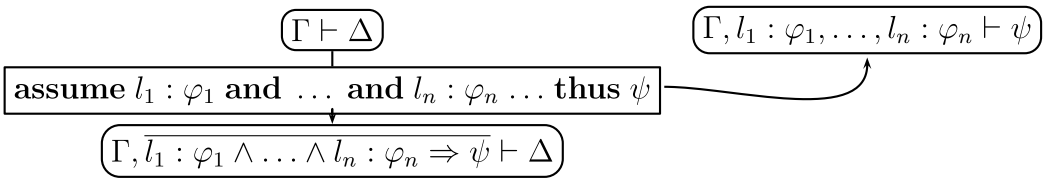
Given a context $\Gamma \vdash \Delta$, the command

$$\textbf{assume } l_1 : \varphi_1 \textbf{ and } \ldots \textbf{and } l_n : \varphi_n \textbf{ thus } \psi \tag{8.6}$$

starts a new proof tree, in which the current context is enriched by the assumption $l_1 : \varphi_1, \ldots, l_n : \varphi_n$:

$$\Gamma, \varphi_1, \ldots, \varphi_n \vdash \psi \tag{8.7}$$

For assumptions, there is no quick test, even though it would be possible to integrate a similar functionality as in MIZAR in case a connection to the current goal can be established. Subsequent operations corresponding to the assumptions are executed on the new proof tree. Thus, the modification of the overall proof plan is as follows:


$\Gamma \vdash \Delta$
assume $l_1 : \varphi_1$ **and** $\ldots$ **and** $l_n : \varphi_n \ldots$ **thus** ψ
$\Gamma, l_1 : \varphi_1, \ldots, l_n : \varphi_n \vdash \psi$
$\Gamma, \overline{l_1 : \varphi_1 \wedge \ldots \wedge l_n : \varphi_n \Rightarrow \psi} \vdash \Delta$

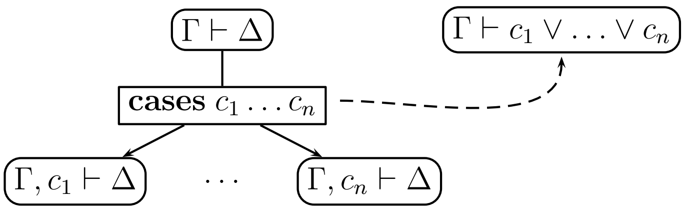
In the proof plan, $\overline{\varphi}$ denotes the closure of φ over new variables that were introduced by the assumptions.

Cases

The command

$$\begin{array}{l} \textbf{case } \varphi_1 \\ \textbf{thus } \psi \\ \quad\vdots \\ \textbf{case } \varphi_n \\ \textbf{thus } \psi \end{array} \tag{8.8}$$

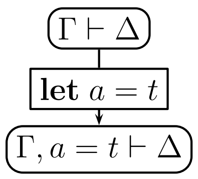
introduces a case distinction over n cases. It reduces the task to n new subtasks, where each subtask has an additional assumption corresponding to the case. The proof obligation associated with the proof command is the condition that the case distinction is exhaustive. All cases must derive the same fact ψ.

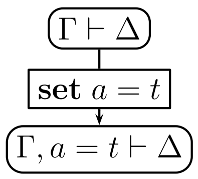

$\Gamma \vdash \Delta$
$\Gamma \vdash c_1 \vee \ldots \vee c_n$
cases $c_1 \ldots c_n$
$\Gamma, c_1 \vdash \Delta$
$\cdots$
$\Gamma, c_n \vdash \Delta$

Quick Test: The quick test tries to close the proof obligation that the case distinction is exhaustive by checking whether there is an axiom or assumption of the form $c_1 \vee \ldots \vee c_n$ (possibly permuted). In the case that the quick test fails, the proof plan is modified as shown below, where the dashed line indicates the proof tree that is introduced when expanding the justification.

Abbreviations and Instantiations

An abbreviation **let** a=t introduces the variable a as an abbreviation for the term t, provided that a does not already occur as a variable in the context. The abbreviation is stored as an additional assumption. The role of abbreviations is to increase the readability of a proof script by reducing its size. Adding an equation $a = t$ with a fresh variable a as premise is conservative in the sense that a proof using the new variable a can be converted in a proof without a by just substituting all occurrences of a by t. Similarly, the command **set** a=t is defined and instantiates an unbound meta-variable in the proof state with the given term t.





There are no quick tests for the **let** and the **set** command.

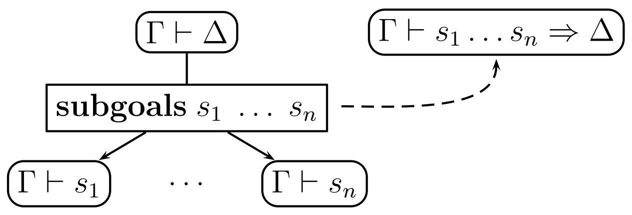
Subgoals:

Given a context $\Gamma \vdash \Delta$, the command

$$\begin{array}{l} \textbf{subgoal } \varphi_1 \\ \quad\vdots \\ \textbf{subgoal } \varphi_n \end{array} \tag{8.9}$$

The command **subgoals** reduces a goal of a given task to n subgoals, each of which is represented as a new task. In case a goal is reduced to a single subgoal, a ; is used as syntactic sugar to avoid the need to start a new subproof. New assumptions can be introduced using the **using** construct.

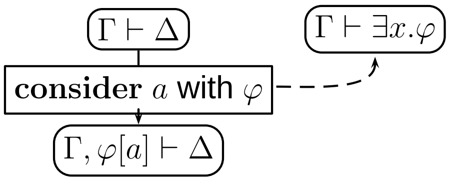
Quick Test: This checks if there is an inference that matches the current goal and has n premises, each of which unifies with one of the specified subgoals.


$\Gamma \vdash \Delta$
$\Gamma \vdash s_1 \ldots s_n \Rightarrow \Delta$
subgoals $s_1 \ \ldots \ s_n$
$\Gamma \vdash s_1$
$\cdots$
$\Gamma \vdash s_n$

If there is no inference introducing the subgoals specified by the user within a single step the repair strategy tries to further reduce the goal in the current task, thus introducing further subgoals, until all specified subgoals are found. If a subgoal matches a specified goal, it is not further refined. If all subgoals are found by a sequence of proof steps, these steps are abstracted to a single justification.

Existential Assumptions

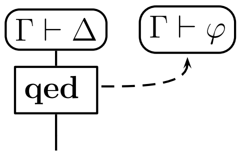
The construct **consider** provides a means to introduce an existential variable based on an existential assumption. It thus corresponds to existential elimination in natural deduction.


$\Gamma \vdash \Delta$
$\Gamma \vdash \exists x.\varphi$
consider a with φ
$\Gamma, \varphi[a] \vdash \Delta$

Quick Test: The quick test checks whether the existential formula occurs as an assumption in the current sequent or is available as an axiom in the current theory.

Qed:

The command **qed** is used to indicate that a task is solved.



Quick Test: The quick test checks whether the goal formula φ occurs in the context, or whether the symbol *false* occurs at top-level on the left hand side of the task, or the symbol *true* occurs at top-level on the right-hand side of a task. A task can also be closed if the inference *name* is applied and all its premises and conclusions are matched to term positions in the current task.

Equational Reasoning and Binary Relations

In addition to reasoning with assertions, many mathematical proofs involve computations in the form of chains of equalities or inequalities. To simulate such equality chains, we provide an abbreviation "." in terms that corresponds to the right-hand side of the previous command, which needs to be a binary predicate. Consider for example the following proof, whose corresponding proof script is shown in Figure 8.6.

Theorem. Every linear function $f(x) = ax + b(a, b \in R, a \neq 0)$ is continuous at every point x_0.

Proof. The claim to be shown is that for every $\epsilon > 0$ there is a $\delta > 0$ such that whenever $|x - x_0| < \delta$, then $|f(x) - f(x_0)| < \epsilon$. Now, since

$$|f(x) - f(x_0)| = |ax + b - (ax_0 + b)| \tag{8.10}$$
$$= |ax - ax_0| \tag{8.11}$$
$$= |a| \cdot |x - x_0| \tag{8.12}$$

it is clear that $|x - x_0| < \epsilon/(|a|)$ implies $|f(x) - f(x_0)| < |a| \cdot \epsilon/(|a|) = \epsilon$. Hence, for all $\epsilon > 0$, $\delta = \epsilon/|a| > 0$ is the number fulfilling the claim.

```
theorem $\exists a, b.a \neq 0 \wedge f(x) = ax + b \Rightarrow cont(f, x_0)$
proof
  $|f(x) - f(x_0)| = |ax + b - (ax_0 + b)|$
  $. = |ax - ax_0|$
  $. = |a| \cdot |x - x_0|$
  assume $|x - x_0| < \epsilon/(|a|)$
    $|f(x) - f(x_0)| < |a| \cdot \epsilon/(|a|)$
    $. = \epsilon$
  thus $|f(x) - f(x_0)| < \epsilon$
  assume $\epsilon > 0$
    let $\delta = \epsilon/|a|$
    $\delta > 0$
  thus $\exists \delta.\delta > 0 \wedge |x - x_0| < \delta \Rightarrow |f(x) - f(x_0)| < \epsilon$
qed
```

Figure 8.6: Realization of the example proof in the proof language

8.4 From Assertion Level proofs to Declarative Proof Scripts

Given a proof plan, we shall show now how we can extract a declarative proof script from it, independently of how the proof plan was generated. In particular, this will allow us to generate a declarative proof script for proofs that have been generated automatically. Note that this is usually not attempted in other proof assistants because of the detailedness of the underlying calculus. For example, in ISABELLE, there "*is currently no way to transform internal system-level representations of Isabelle proofs back to Isar text*" (see [Wen99b], p. 11). However, within our setting, the assertion level builds the lowest layer of abstraction, and a presentation is therefore reasonable. Indeed, the assertion level was motivated and developed originally for a human oriented proof presentation. However, let us point out that even though the assertion level proofs are rather abstract, one might want to transform the generated proof slightly for the presentation purpose, as the way a proof is found is often different from the way it is presented in a textbook:

> *In general it is not practical to write the entire thought process that goes into a proof, for this requires too much time, effort, and space. Rather, a highly condensed version is usually presented and often makes little or no references to the backward process. [...] There are several reasons why reading a condensed proof is challenging:*
>
> 1. *The steps of the proof are not always presented in the same order in which they were performed when the proof was found.*
> 2. *The names of the techniques are often omitted.*
> 3. *Several steps of the proof are often combined into a single statement with little or no explanation.*

(from [Sol05] p. 13-15)

In the sequel, we present an algorithm that transforms a proof tree into a declarative proof script, thereby taking into account that backward steps are often converted into forward steps. Indeed, forward style proofs are usually more natural than backward style proofs (see for example [Wen99a]). Note however that our translation represents just one possibility and that other translating/rendering functions can easily be defined.

The second aspect, namely "condensing" of proof scripts is not considered here (except for a trivial case) and can be seen as a research field in its own right. One possibility, which has been worked out, is the use of expert knowledge for the classification task (see [SB09a, SB09c, SB09b, Sch10]). Interestingly, the expert knowledge can even automatically be extracted from annotated proof examples via machine learning techniques.

Let us recall that an assertion level step reduces a task T to a set of subtasks $T'_1, \dots, T'_n$. We can analyze such a reduction in order to compute a corresponding proof command corresponding to that step. We assume a function **label** which returns the set of those labels which are used in premises and conclusions of the proof operator and "**.** " if none of them has a label.

Definition 8.4.1 (Task Difference)**.** *Let T, T' be sequents. The difference between T and T' is defined as follows:*

$$\textbf{diff}\ (T, T') = \langle \{\varphi \in \Gamma' | \varphi \notin \Gamma\}, \{\xi \in \Delta' | \xi \notin \Delta\} \rangle \tag{8.13}$$

Based on the task differences of a reduction step, it is possible to define a translation function $\mathcal{G}[\![\cdot]\!]$ which takes a task T (more precisely a node of the task tree) as input and returns a proof script. To make the classification of proof steps explicit, we explicitly label forward steps by **have**. To keep the proof script natural, we assume single conclusion sequents. This can be achieved in classical logic by modifying inference rules that introduce a multiple conclusion by a variant that introduces negated assumptions and a single new conclusion.

The translation function $\mathcal{G}[\![\cdot]\!]$ is defined with the following cases:

Non branching rules:

- The step solves the goal. In this case, the empty string ϵ is inserted and the proof is completed.
$$\mathcal{G}[\![T]\!] = \epsilon \text{ provided that } \operatorname{succ}(T) = \emptyset \tag{8.14}$$
- The step introduces a new subgoal and leaves the assumptions unchanged. In this case, the backward step is converted to a forward step and the translation function recursively invoked on the subproof. An example is shown below:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash X \subset A}}{\Gamma \vdash X \in \mathcal{P}(A)}$$

have $L_1 : X \subset A$
proof
$\vdots$
qed
have $L_2 : X \in \mathcal{P}(A)$

Formally:

$$\mathcal{G}[\![T]\!] = \textbf{have } L_1 : \varphi \textbf{ proof } \mathcal{G}[\![T_1]\!] \textbf{ qed have } \xi \textbf{ from } L_1$$
$$\text{provided that } T = \Gamma \vdash \xi, \operatorname{succ}(T) = \{T_1\}, \textbf{diff}\ (T, T_1) = \langle \emptyset, \{\varphi\} \rangle \tag{8.15}$$

- The step introduces new assumptions, but leaves the goal unchanged. In this case, each new assumption gives rise to a new proof step that is classified as **have**. The translation function is recursively invoked on the successor task. Note that it would also be possible to extend a forward step such that several assumptions can be introduced simultaneously, in analog to the **assume** construct. This would be a solution if one would be interested in an invariant that a single proof step always corresponds to a single proof command. However, we found the proof scripts resulting with the first option more natural.

$$\frac{\Gamma, x \in A, x \in B \vdash \varphi}{\Gamma, A \cap B \vdash \varphi}$$

have $L_1 \!: x \in A$
have $L_2 \!: x \in B$
⋮

Formally:

$$\mathcal{G}[\![T]\!] = \textbf{have } L_1 : \varphi \ \ldots \ \textbf{have } L_n : \varphi_n \ \mathcal{G}[\![T_1]\!]$$
$$\text{provided that; succ}(T) = \{T_1\}, \textbf{diff } (T, T_1) = \langle \{\varphi_1, \ldots, \varphi_n\}, \emptyset \rangle \quad (8.16)$$

- The step introduces a subgoal together with new assumptions. In this case, the step is mapped to an **assume** step, whose inner part is provided by the recursive call of the translation function.

$$\frac{\Gamma, x \in A \vdash x \in B}{\Gamma \vdash A \subset B}$$

assume $L_1 \!: x \in A$
⋮
thus $L_2 \!: x \in B$

Formally:

$$\mathcal{G}[\![T]\!] = \textbf{assume } L_1 : \varphi_1 \textbf{ and } \ldots \textbf{and } L_n : \varphi_n \ \mathcal{G}[\![T_1]\!] \textbf{ thus } L_{n+1}\psi$$
$$\text{provided that; succ}(T) = \{T_1\}, \textbf{diff } (T, T_1) = \langle \{\varphi_1, \ldots, \varphi_n\}, \{\psi\} \rangle \quad (8.17)$$

Let us remark that for non-branching rules all possibilities are covered.

Branching rules:

For branching rules, we consider the following cases:

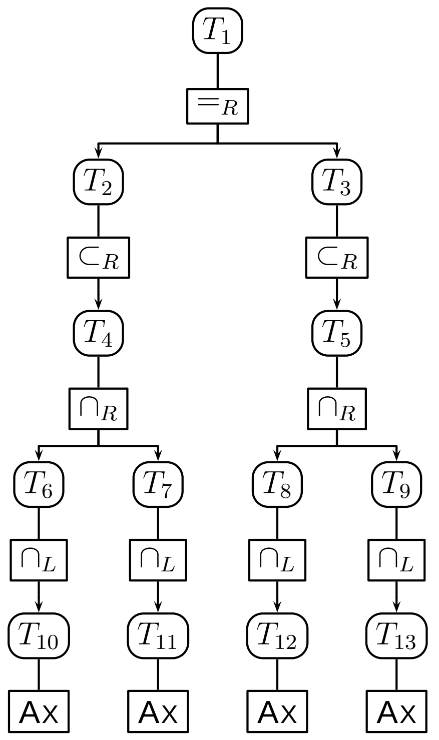
- The step reduces the goal to several subgoals with new assumptions, but the subgoals are identical. This is for example the case when using the assertion

$$\frac{\Gamma, x \in A \vdash \varphi \qquad \Gamma, x \in B \vdash \varphi}{\Gamma, x \in A \cup B \vdash \varphi}$$

case $L_1 \!: x \in A$
⋮
case $L_2 \!: x \in B$
⋮
thus φ

Formally:

$$\begin{aligned}\mathcal{G}[\![T]\!] = &\textbf{case } L_1 : \varphi_{11} \textbf{ and } \ldots \textbf{and } \varphi_{1m_1} \mathcal{G}[\![T_1]\!] \textbf{ thus } \phi \\ &\vdots \\ &\textbf{case } L_n : \varphi_{n1} \textbf{ and } \ldots \textbf{and } \varphi_{nm_n} \mathcal{G}[\![T_n]\!] \textbf{ thus } \phi \\ &\qquad \text{provided that } \text{succ}(T) = T_1, \ldots, T_n, n > 1 \\ &\qquad \textbf{diff } (T, T_i) = \langle \varphi_{i1}, \ldots, \varphi_{im_i}\}, \{\psi\} \rangle \qquad (8.18)\end{aligned}$$



T_1: $\vdash A \cap B = B \cap A$
T_2: $\vdash A \cap B \subset B \cap A$
T_3: $\vdash B \cap A \subset A \cap B$
T_4: $x \in A \cap B \vdash x \in B \cap A$
T_5: $x \in B \cap A \vdash x \in A \cap B$
T_6: $x \in A \cap B \vdash x \in A$
T_7: $x \in A \cap B \vdash x \in B$
T_8: $x \in B \cap A \vdash x \in B$
T_9: $x \in B \cap A \vdash x \in A$
T_{10}: $x \in A, x \in B \vdash x \in A$
T_{11}: $x \in A, x \in B \vdash x \in B$
T_{12}: $x \in A, x \in B \vdash x \in B$
T_{13}: $x \in A, x \in B \vdash x \in A$

Figure 8.7: An automatically generated proof tree

- Otherwise we consider each difference separately and reduce it either to an assumption or to a forward step, according to the non-branching rules.

$$\frac{\Gamma \vdash x \in A \qquad \Gamma \vdash x \in B}{\Gamma \vdash x \in A \cap B}$$

have *L1*:$x \in A$
have *L2*:$x \in B$
have *L3*:$x \in A \cap B$ **from** L1,L2

Formally

$$\mathcal{G}[\![T]\!] = L1 : \ \mathcal{G}[\![T_1]\!] \ \ldots \ L_n : \mathcal{G}[\![T_n]\!] \ \textbf{have} \ \psi \ \textbf{from} L1, \ldots, L_n$$
$$\text{provided that } \operatorname{succ}(T) = T_1, \ldots, T_n \quad (8.19)$$

Simplifications

Moreover, we use the following simplification rule to remove one step subproofs (which we consider to be simple):

have L_1:A
proof
  have L_2:B **from** N $\longrightarrow$ **have** L_1:A **from** N
N
qed

Note that this step corresponds to the use of the close-direction of the assertion instead of a forward direction followed by the axiom rule. Thus, when allowed and preferred during the proof search, the above transformation is not necessary.

```
Theorem SET013+4:  $A \cap B = B \cap A$
proof
  L1:  $A \cap B \subset B \cap A$
  proof
    assume L2:  $X \in A \cap B$
      L3:  $X \in B$ from L2
      L6:  $X \in A$ from L2
    thus $X \in B \cap A$
  qed
  L9:  $B \cap A \subset A \cap B$
  proof
    assume L10:  $X \in B \cap A$
      L11:  $X \in A$ from L10
      L14:  $X \in B$ from L10
    thus $X \in A \cap B$
  qed
  $A \cap B = B \cap A$ from L1,L9
qed
```

Figure 8.8: Resulting declarative proof script for TPTP problem SET013+4

8.4.1 Examples

To get a feeling for the structure of the generated proof scripts, we give two examples of declarative proof scripts that have been automatically generated. Consider the generated proof tree, shown in Figure 8.7, proving the TPTP problem SET013+4 produced by the assertion level prover (which will be presented in Chapter 12).

The generated proof script is shown in Figure 8.8. Similarly, producing the proof from the proof of problem SET015+4, we obtain the proof script shown in Figure 8.9. Notice the handling of the disjunction in the subproof of L3: Instead of introducing two literals on the right-hand side, the first literal is negated and moved to the left-hand side of the sequent. The same operation is performed in the subproof of L7.

Let us remark that our approach can easily be extended to the case where quantifiers are explicitly contained in formulas. Then, one would refine the non-branching rules of $\mathcal{G}[\![\cdot]\!]$ to include a command corresponding to $\forall_I$, such as **fix**.

8.5 Related Work

Central to the work described in this section is the notion of a proof plan and its relation to a declarative proof script. As possible choices to support meta-variables have already been discussed in Section 8.2.1, we focus here on the more general picture.

8.5.1 Underspecified Proof Scripts

For Mizar, Wiedjik introduces the notion of a proof sketch, which is a declarative proof script with gaps: "*A formal proof sketch is a completely correct Mizar text, apart from errors *4 and *1. These errors say that reasoning steps are not justified*" (see [Wie04] page 3). Several systems have been developed that try to automatically close such gaps, such as Mizar, Nqthm [BM88], the Spl system [Zam99], the Sad system [VLP07], the Naproche [KCKS09] system, the Scunak system [Bro06], as well as Tutch [ACP01].

```
Theorem SET014+4 $A \cup B = B \cup A$
proof
  L1:  $A \cup B \subset B \cup A$
  proof
    assume L2:  $X \in A \cup B$
      L3:  $X \in B \vee X \in A$
      proof
        assume L4:  $X \notin B$
          case $X \in A$
          thus $X \in A$
          case $X \in B$
          thus $X \in A$
        thus $X \in A$
      qed
    thus $X \in B \cup A$ from L3
  qed
  L5:  $B \cup A \subset A \cup B$
  proof
    assume L6:  $X \in B \cup A$
      L7:  $X \in A \vee X \in B$
      proof
        assume L8:  $X \notin A$
          case $X \in B$
          thus $X \in B$
          case $X \in A$
          thus $X \in B$
        thus $X \in B$
      qed
      hence $X \in A \cup B$ from L7
    thus $X \in A \cup B$
  qed
  hence $A \cup B = B \cup A$ from L1,L5
qed
```

Figure 8.9: Declarative proof script generated for the TPTP problem SET015+4

Here, a structure which can be understood as a proof plan is constructed from a textbook proof, rather than automatically generated by a proof planner. However, the proof plan is not always explicit in the form of a data structure.

The relationship between explicit proof plans and proof sketches was already exploited in the previous ΩMEGA system, which provided the island tactic "to insert arbitrarily large gaps in the proof" (from [SBF^{+}03] page 18). These were later justified by expanding the island step. However, the relationship to declarative proof languages to specify island steps was not exploited.

ISAPLANNER [DJ07], a proof planner built on top of ISABELLE, exploits the connection between proof plans and declarative proof scripts even further. A key feature of ISAPLANNER is that proof plans are equivalent to proof scripts. More specifically, proof plans are build by abstract elements that correspond to ISAR commands and that have been lifted to the abstract level of proof plans. Abstract elements have a name, such as apply, the arguments corresponding to the element, an execution function, and a pretty printer. Among others, ISAPLANNER provides the abstract element **gap**, which takes as optional argument a proof planning technique to close the gap, and an execution function that skips the proof using an oracle. Proof plans are constructed based on reasoning techniques that are executed on reasoning states. A reasoning state contains contextual information, such as Rippling annotations, as well as a continuation describing the next reasoning technique to be applied. In contrast to ISABELLE, ISAPLANNER supports meta-variables that occur in assumptions, which are not allowed when writing proofs in ISAR. This is possible by giving stable references to assumptions, goals and meta-variables in the form unique names. Internally, ISAPLANNER stores meta-variables in a table which is indexed by their unique name and which holds the names of goals, assumptions and other meta-variables in which they occur. It is important to note that only tactics that have been lifted to the level of reasoning techniques can be used to construct a proof plan, in particular, it is not possible to reconstruct a proof plan/proof script from an internal proof.

8.5.2 Proof Script Extraction

There exist also several approaches to present a machine-found proof in a user friendly way [Hua96, Fie01]. In [Sac10] a language is presented to automatically generate declarative proofs from proof terms. While this allows the presentation of proofs which have been found automatically, the resulting proofs are very detailed. Within our setting, the quality of the proofs benefit from the abstract structure of the generated proof tree, which is at the assertion level (or even above). The importance of the source proof for proof presentation has already been recognized by Felty and Miller twenty years ago: "*Since the mechanism for translating a proof tree into text is simple, much of the challenge in constructing natural text can be transferred to constructing proof trees: to first generate good text, generate good proof terms*" (see [FM88] p. 4).

8.5.3 Declarative Proof Languages

The AUTOMATH project [dB70] was the first significant attempt to formalize mathematics and to use a computer to check its correctness. AUTOMATH was a pure proof checker with a rather cryptic input language that did not provide tools for proof automation. Subsequently, several research projects dealt with its proof automation as well as increasing the readability of the input text. While the derivatives of AUTOMATH, Mathematical Vernacular[dB94], weak type theory [Ned02], or MATHLANG [KMW04] are close to natural language, many other (formal) proof languages similar to ours have been developed. We subsequently compare our proof language with both the MIZAR and the ISAR language, which are probably most similar to our language.

Mizar: MIZAR was the first proof assistant that implements what is today known as the declarative style of proof. MIZAR is a batch proof assistant that is not interactive. Rather, a file is loaded and error messages written in the body of the input text. In particular, MIZAR continues to check after a step could not be justified. To justify a step, MIZAR invokes an internal first order prover that works with normal forms. This means that a direct back-translation of the generated proof is not possible. Moreover, the automation facilities of the system cannot be extended by the user. In contrast to our approach, MIZAR does not construct a proof plan representation, but checks the steps immediately. Parallel checking is currently not implemented in MIZAR.

Isar: In ISAR, a proof consists of a claim followed by a proof according to ISAR's formal syntax. Similar to our language, ISAR provides justification hints in the form of two constructs **by** and **from**. This means that the automation facilities can be extended by the user. However, in contrast to our approach, the order of facts is used to speed up the checking process. Gaps are not natively supported by ISAR, and the checking cannot be continued after the first error, even though this could be realized easily. In contrast to our approach, ISAR does not feature meta-variables. Moreover, it is not possible to translate back an internal proof to a declarative proof script, which is a key feature of our approach.

8.6 Summary

In this chapter we introduced a declarative proof language and illustrated the connection between declarative proof scripts and proof plans: We showed how a declarative proof script can be used to construct an initial proof plan, which is subsequently refined to obtain a formal proof. Conversely, we showed how a declarative proof script can be extracted from a proof plan by means of a PDS-view that selects one specific proof alternative (Contribution A1(vi), Section 1.1). While this extraction process might generally produce proofs that are too detailed, we showed that proofs at the assertion levels are well-suited for such a translation due to their abstract granularity.

9 Heuristic Control and Compilation of Inferences

Deep inference deduction systems remove the restriction that inference rules can only be applied to top-level formulas and instead they allow their application to subformulas, just as in most rewriting systems. This has the advantage that shorter proofs can be constructed, but comes along with some disadvantages: (i) The branching factor increases. (ii) The complexity of the matching process increases, as more matching candidates need to be considered. (iii) The number of redundancies in the search space increases, as there are more possibilities to derive a particular statement. (iv) The resulting proof steps might become difficult to understand.

Therefore, for practical applications one has to consider the tradeoff between shorter proofs and higher nondeterminism in the search space and hence needs a better control over the depth of the inference application. In this section, we introduce a specification language to attach user defined control information to inferences to restrict their applicability and to influence their effects. The annotations enable a high-level and fine-grained though declarative control of the internal search procedure, supporting the full range from top level inference to full deep inference, as well as anything in between. In order to obtain reasonably efficient proof search procedures comparable to built-in procedures, compilation techniques for inferences are developed. Moreover, compilation provides an additional abstraction layer which can easily be extended and optimized.

9.1 Dynamic Effects and User-Defined Constraints

Generally, a single inference can be applied in many ways (see Chapter 6), but not all of them will necessarily contribute to the goal of the current proof. Rather, efficient (and controlled) search is achieved by choosing an appropriate subset of the many application directions, as well as by providing mechanisms to avoid redundancies in the search space.

The problem of controlling the application of inferences has already been studied in the context of *proof planning methods* in the previous ΩMEGA system (see for example [MS99a, Ker98] for a detailed description). Premises and conclusions of methods were annotated with $\oplus$ and $\ominus$ to indicate a subset to be matched for the method to be applicable.

Moreover, facts marked by $\ominus$ were removed from the proof state to avoid redundancies. The remaining choice points were controlled by control rules.

Another view is to consider the application of a set of inferences as a classical retrieval problems of *candidate terms* from a given *query term* or *subject term* satisfying a specific relationship. Indexing data structures have been developed to support such operations efficiently for automated reasoning. However, within our setting, there are the following differences with respect to this:

- The instantiation of a single inference can be seen as a simultaneous unification/-matching problem, as several premises/conclusions can be instantiated. Simultaneous matching/unification has been studied for hyperresolution, where one has to solve the problem of simultaneous retrieval of unifiable terms. Instead of matching all inference nodes simultaneously, a reasonable strategy is to match one inference node at a time, starting from an inference where no node has been matched. This has been done for example in VAMPIRE [RV99], where the retrieval problem for n terms is reduced to n retrieval problems of one term. Techniques for simultaneous unification are usually much more complicated than for a single term and known as *multiterm indexing* (see [RSV01]).

- In addition to the unifier, the proof obligations that arise from negative positions need to be determined. These proof obligations are dynamic and context dependent.

- Instead of using one static relation that expresses the relationship the candidate terms have to satisfy, we want to support arbitrary, user-defined constraints that are locally defined with respect to an inference node.

- Additional search control, such as backtracking or deletion of a formula after its use are tasks that have to be accomplished independently of the retrieval.

Finally, we can see the deep application of a rule as a general form of rewriting. To that end, let us consider the application of a single resolution replacement rule in more detail. We observe that its effect corresponds to the result of a tree replacement. Thus, the overall process, matching and execution, can be split into two operations, namely (i) to find all possible matches within a tree, and (ii) to select a candidate on which a particular action is performed. These two operations are known in the rewriting community as *tree pattern matching* and *reduction strategy*. In the context of rewriting, traversal functions provide a simple mechanism to make the selection of the redex explicit and the overall rewriting strategy configurable. Moreover, efficiency at runtime is often obtained by compiling patterns to programs (see for example [BM06, vdBHKO02, Vit96, MK98, MRV01]), which results in significant speed-ups of the execution time as compared to interpreters.

We will follow this approach in the sequel. First, we introduce a language to annotate inferences with control information. We then show how annotated inferences can be compiled to programs which are then executed. Compilation will speed up the matching process, in particular because it avoids the passing of bindings, and because we can optimize programs such that control information is only locally evaluated. From a conceptual point of view, speaking about programs will introduce a convenient abstraction layer.

Compilation is the technique of our choice, because it allows the incorporation of user-defined constraints in a convenient way. Moreover, we are especially interested in applications in which the set of patterns, given, e.g., by a strategy, remains fixed, and we can therefore distinguish between preprocessing time and matching time. Minimizing matching time is the first priority, as preprocessing can often be done once and for all and the result be stored in a separate file. Exploiting the idea to reduce a simultaneous matching problem to a sequence of subsequent matching problems, we can extend this to inferences that require several matches. Using this relationship allows us to adapt known techniques to our setting.

Before defining the annotation language, we present some examples of the search restrictions we want to be able to express.

Restriction of the Application Directions

Suppose the current task is to show

$$A \subseteq B, x \in B \Rightarrow x \in A \vdash A = B \tag{9.1}$$

and we are given the inference rule

$$\frac{P_1 : A \subseteq B \quad P_2 : B \subseteq A}{C : A = B} \tag{9.2}$$

originating from the assertion $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$.

Suppose further that the proof requires to unfold the definitions in the goal first and then to use logical arguments to finish the proof. With respect to the strategic goal, only those application directions of this inference make sense, in which the conclusion C is instantiated, i.e., the following four PAIs: $\{P_1, P_2, C\}, \{P_1, C\}, \{P_2, C\}, \{C\}$.

Instead of always enumerating all desired inference applications, it is often simpler to specify a subset of premises and conclusions that must or must not be instantiated, for example, to use only those directions which do not introduce new meta-variables. In addition to stating a main direction for the inference, such as *forward*, *backward*, and *close*, we want also to specify for a single inference node whether it must be instantiated or not. Note that this kind of restriction is static, i.e., the question whether a certain direction is allowed or not does not depend on the current proof situation, but is decided in advance. An example is given below, where we use the * to indicate that a particular inference node must be instantiated:

$$\frac{P_1 : A \subseteq B \quad P_2 : B \subseteq A}{C : A = B\{*\}} \tag{9.3}$$

Contraction/Weakening of Assumptions

In addition to the specification of a set of application directions, we want to be able to specify whether a used formula should be removed from the proof state after the application of the inference. For example, when unfolding a definition, it is often reasonable to remove the unexpanded definition from the proof state (i) to avoid loops, and (ii) to avoid unnecessary matching attempts. Consider for example the task

$$x \in A \cup B \vdash x \in B \cup A \tag{9.4}$$

Expanding the definition in the antecedent adds the new premise $x \in A \vee x \in B$. There is no need to keep the fact $x \in A \cup B$, as the task is provable iff

$$x \in A \vee x \in B \vdash x \in B \cup A \tag{9.5}$$

is provable. However, as an alternative, consider the task

$$A \subset B, x \in A, y \in A \vdash x \in B \wedge y \in B \tag{9.6}$$

and the inference

$$\subseteq \frac{P_1 : U \subseteq V \quad P_2 : x \in U}{C : x \in V} \; x \text{ new for } U \text{ and } V \tag{9.7}$$

which we want to apply in the forward direction. Without performing a manual contraction step on $A \subset B$, it is not reasonable to remove the fact $A \subset B$, as we may need several forward instances of it to derive all necessary facts.

We use the annotation `-` to indicate that a used fact is to be removed:

$$\frac{x \in A \cup B\{\texttt{-}\}}{x \in A \vee x \in B} \tag{9.8}$$

Avoiding redundancies is a key step to implement efficient automated theorem provers. In clausal-based theorem provers, this is connected to the question to delete clauses under certain circumstances. Intuitively, clauses can be deleted if they are tautologies or if they are subsumed by other clauses.

Backtracking Behavior

Another kind of redundancy that can occur is due to derivations which are rotational variants of each other and derive the same fact. In the case of backtracking, there is no benefit to explore these alternatives, as they would lead to the same result and trigger backtracking again. Consider the task

$$\Gamma, x \in A \cup B, x \in B \cup C \vdash \Delta \tag{9.9}$$

Expanding the first occurrence of $\cup$ and then the second occurrence has the same effect as expanding first the second and then the first. When computing all possibilities to apply the definition of $\cup$, both alternatives are computed and the second stored for backtracking. Note that in large proofs many such rotations are possible and should be eliminated. This is possible by providing a possibility to control the number of alternatives that are computed when instantiating the inference. In the example above, it is reasonable not to compute every alternative, as the above transformations are equivalence transformations. Similarly, if we consider for example a confluent terminating rewrite system and have found an applicable rewrite rule at a preferred position, it is usually reasonable to apply the rule directly instead of looking for further applicable rules, since backtracking is never required for such systems of rules.

Order of the Query Terms

When reducing the inference-matching/unification problem to a sequence of individual matching problems, the order in which the query is executed matters. Consider for example the inference

$$\frac{x \in A\{*\} \quad A \subset B\{*\}}{x \in B} \tag{9.10}$$

and the task

$$x \in U, y \in V \vdash x \in U \cap V \tag{9.11}$$

Starting the query with $A \subset B$ immediately stops the overall query because a suitable partner cannot be found in the sequent. Therefore, in this example, search can be avoided when using a particular order. Generally, it is often better to match large terms first, as they are more unlikely to be found in a given task. Moreover, they constrain subsequent queries in the case of overlapping variables. Instead of providing annotations to influence this order, we keep its significance in mind and aim at an automatic solution for it.

Deep vs. Shallow Inference

Extending the application of an inference from top level formulas to subformulas increases the number of candidate terms a premise/conclusion can be matched against. This increases the branching factor, but might also result in proof steps which are difficult to understand. Depending on the setting, it is therefore desirable to have a finer control on this matching process. To be able to cover the full range from shallow inference to deep inference, we provide the annotation $[\cdot]$ to enable deep inference, respectively to switch it off. Consider the task

$$\vdash x \in A \cup B \vee x \in B \cup A \tag{9.12}$$

and the two inferences

$$\frac{x \in A \vee x \in B}{x \in A \cup B} \qquad\qquad \frac{x \in A \vee x \in B}{[x \in A \cup B]}$$

While the premises and conclusion of the first inference can only be matched against the top-level formulas of the sequent, the conclusion of the second inference carries the annotation $[\cdot]$, which enables the deep matching. Therefore, only the second of both inference can be applied.

Absolute Position Restrictions

Another restriction controls the candidate formulas which can be used for the matching. For heuristic considerations, this is essential: The right hand side of the sequent indicates the main goal we are trying to solve and thus represents important information which we might want to exploit within a proof strategy. Treating all positive subformulas as goal is not an option for strategic proof search. Consider for example the sequent

$$\left(S^+ \wedge Q^+\right)^+ \Rightarrow \left(P^- \wedge Q^-\right)^-, \left(A^+ \wedge B^-\right)^+ \Rightarrow S^- \vdash \left(P^+ \wedge Q^+\right)^+ \tag{9.13}$$

To obtain a goal directed search strategy, one should give the positive goal formula $(P^+ \wedge Q^+)$ a preference over other positive formulas. This has already been realized e.g. by Sieg [SB98, SRL$^+$06, Sie09] and is used by the *intercallation calculus*: The intercallation calculus is a restriction of natural deduction, where normality of the proof is enforced by applying the elimination rules only on the left to premises and the introduction rules only on the right to the goal. In the first case one really tries to *extract* a goal formula by a sequence of E-rules from an assumption in which it is contained as a strictly positive subformula.

> *This feature is distinctive and makes search efficient, but it is in a certain sense just a natural systematization and logical deepening of the familiar forward and backward argumentation.* *(from [Sie09] p. 11)*

Therefore, we provide the possibility to restrict candidate terms of a premise/conclusion by a declarative sequent matcher:

$$\frac{[P]}{P\{\vdash *\}} \tag{9.14}$$

Dynamic Position Restrictions

We cannot only impose absolute position restrictions, as above, but also dynamic position restrictions. Consider the following proof situation

$$\left(S^-_{(1)} \wedge \left(Q^+ \Rightarrow S^-_{(2)}\right)\right)^- \Rightarrow S^+_{(3)} \tag{9.15}$$

where (1), (2), and (3) denote the occurrences and deep axiom rule

$$\frac{[P]}{[P]} \tag{9.16}$$

There are two negative occurrences of S ($S_{(1)}$ and $S_{(2)}$) as well as a positive occurrence of S ($S_{(3)}$). Thus, there are two application possibilities of the deep axiom rule. However, they differ as follows: While the use of $S_{(1)}$ is for free, the use of $S_{(2)}$ introduces the proof obligation P. The difference is due to their relative position to $S^+_{(3)}$: While for $S_{(1)}$ there is no β-formula on the path to $S_{(3)}$, there is one when considering $S_{(2)}$, giving rise to the proof obligation. Therefore, we provide an annotation `nopob` that restricts premise candidates to those which do not introduce new proof obligations, as indicated below:

$$\frac{[P]\{nopob\}}{[P]} \tag{9.17}$$

Meta-level Restrictions

Often, a simple static analysis helps to drastically reduce the number of choice points. In the context of rewriting, it is often possible to orient an equation with respect to a given ordering and performing rewriting only in one direction. However, it is easy to construct examples of equations which cannot be directed.

Example 9.1.1. *A simple example is a permutative axiom like the commutativity of addition*

$$x + y = y + x \tag{9.18}$$

Other equations which cannot be directed are equations which do not have the same variables on the left and right-hand side, such as

$$f(x) = g(y) \tag{9.19}$$

Rather than disallowing these equations, a possible solution consists of moving the reduction test to runtime by allowing the application of rewrites only if the instance can be oriented. For example, using the lexicographic path ordering, the following instance of (9.18) can be directed:

$$x + f(x) = f(x) + x \tag{9.20}$$

Performing the check at runtime allows the use of the equation in both directions, however, at the cost of the additional check. This technique is known as *ordered rewriting*, which can be seen as a special case of *constrained rewriting*.

Individual Matching

The most basic and at the same time most expensive operation when searching for PAIs is unification. Hence it is crucial to be able to influence unification as best one can. In general, we are in a higher order setting, and higher-order unification will be needed. However, there might be situations in which we are only interested in first order unifiers or even in first order matchers. Consider the following inference representing the induction axiom for natural numbers,

$$\frac{P1: P(0) \quad \begin{array}{c}[P(y)]\\ \vdots \\ P2: P(s(y))\end{array}}{C: P(x)} \tag{9.21}$$

and suppose that the proof situation is to show

$$x + y = y + x \tag{9.22}$$

Higher-order unification produces the following unifiers for P:

$$\begin{aligned}
P &\mapsto \lambda z.z = y + x\\
P &\mapsto \lambda z.z + y = y + z\\
P &\mapsto \lambda z.z + y = y + x\\
P &\mapsto \lambda z.x + z = z + x\\
P &\mapsto \lambda z.x + z = y + x\\
P &\mapsto \lambda z.x + y = z\\
P &\mapsto \lambda z.x + y = z + x\\
P &\mapsto \lambda z.x + y = y + z
\end{aligned}$$

Only the second and the forth solutions are desired; all the other unifiers do not contribute to the solution, even though they are syntactically correct. The solution here is to define a special matcher, taking a formula Φ and producing for each δ-variable v^δ in Φ the unifier

$$P \mapsto \lambda z.\Phi[z/v^\delta]$$

We can also imagine to define a unification algorithm for theory unification (see [BS01a]) or even to cooperate with a computer algebra system to generate a unifier/matcher. There is also the interesting field of so-called algebraic matching algorithms which allow for algebraic manipulations of terms to match terms and provide results such as $x = \sqrt{2}$.

Preference Redex

We would also like to express preferences among positions to further control the order in which the search space is unfolded. From the area of rewriting different reduction strategies are known and implemented by the choice of the traversal order. Examples are leftmost/rightmost, innermost/outermost, innermost parallel, and outermost parallel rewriting. These result in different derivations, as illustrated below:

$$square(3 + 4) \rightarrow square(7) \rightarrow 7 * 7 \rightarrow 49 \tag{9.23}$$

$$square(3 + 4) \rightarrow 3 + 4 * 3 + 4 \rightarrow 7 * 3 + 4 \rightarrow 7 * 7 \rightarrow 49 \tag{9.24}$$

We want to support a similar functionalities to control matching of inference nodes.

9.2 Annotated Inferences

We will now define a declarative syntax for inferences that allows the specification of inferences as well as restrictions on the search process. Figure 9.1 shows the declarative syntax to describe inferences:

```
<inference> ::= (<name> <eqinf> | <inf>) (where <cond>)?
<inf>       ::= <prems> ==> <con>+
<eqinf>     ::= <prems> <term> == <term>
<prems>     ::= <prem> | <prem> ; <prems>
<prem>      ::= ([<forms>] ..)? <form>
<con>       ::= <form> | [<form>]
<termnode>::= <term>
<cond>      ::= <pred>+
<form>      ::= <name>: <term>
<form>      ::= <form> | <form> , <forms>
```

Figure 9.1: Basic syntax to define inferences

Having the general form of inferences from Figure 9.1 in mind, the syntax is best explained by examples. We adopt the usual convention to use capital letters to denote meta-variables and lower-case letters to match constants of the domain, existing variables or new Eigenvariables.

- `axiom: P ==> P` specifies the axiom rule;
- `implI: [F] .. P:G ==> C:F => G` specifies an inference with a conclusion `F => G` named `C` and a single premise `G` named `P` and hypotheses `F`;
- `NatInduction: base:P(0) [P(y)] .. step:P(suc(y)) ==> P(x)` **where** `new(y)` specifies the induction rule for natural numbers naming the cases `base` and `step` respectively and requiring the Eigenvariable condition for y.

Please note that each inference corresponds to a formula in Core's indexed formula tree. This means that quantifiers for free variables need to be reconstructed before the insertion of the formula. Eigenvariables are introduced by a condition `new(`x`;`$y_1, \ldots, y_n$`)`, indicating that x is an Eigenvariable that must be new with respect to $y_1, \ldots, y_n$, i.e., must not occur in $y_1, \ldots, y_n$. In the case that the inference is generated from an axiom, such a specification is not necessary, as all conditions are already determined by the axiom.

Let us now extend the basic inference language by further annotations, as shown in Figure 9.2. The basic language is the same, but premises and conclusions are replaced by annotated versions <aprem> and <acon>, respectively.

In order to enable the application of an inference, candidates for premises, respectively the conclusion are searched in a sequent. By default, for a premise or conclusion, we search inside all formulas of the sequent for subformulas that unify with the given formula. The annotations < annot > controlling that search are thus attached to each premise and conclusion, respectively, and allow to specify (i) whether or not a partner subformula must be found and how it is identified (<occ>), (ii) if candidates should be searched in subformulas and how the search traverses the formulas (<traversal>), and (iii) when the search for a partner shall be aborted (<abort>). More specifically `*` and `-` for a premise or conclusion indicate that a partner must be found for it; in case of `*` the matching

<aprem>	::= <aform> | [<form>] .. <aform>
<aform>	::= <form> {<annots>}? | [<form>] {<deepannots>}?
<deepannots>	::= <annots>? {<restrict>}?
<restrict>	::= `only-left` | `only-right`
<annots>	::= <annot> | <annot>, <annots>
<annot>	::= <traversal> | <occ> | <abort> | nopob |<pred> | <pos>
<occ>	::= `check=`<alg> | + | `!` | `-`
<abort>	::= `abort=`<abort>
<traversal>	::= <lazyall> | <lazyinnerouter> | <first>
<lazyall>	::= <direction>,<lazyprogress>
<lazyinnerouter>	::= <innerouter>,<lazyprogress>
<first>	::= <innerouter>, <leftright>
<direction>	::= `Down` | `Up`
<innerouter>	::= `innermost` | `outermost`
<lazyprogress>	::= `LazyLeft` | `LazyRight`
<leftright>	::= `leftmost` | `rightmost`
<pos>	::= <formulas> (,*)? `|-` <formulas> (,*)?

Figure 9.2: Syntax for Annotated Inferences

formula is kept upon inference application while `-` requires the matched formula to be replaced. For instance, the alternatively annotated inferences

```
implI1: [F] .. P:G ==> C:F => G {*}
implI2: [F] .. P:G ==> C:F => G {-}
```

both require to match `F => G` but only the second also requires to remove it upon application. As an effect they give result respectively in the following derivations:

$$\frac{\Gamma, F \vdash G, F \Rightarrow G, \Delta}{\Gamma \vdash F \Rightarrow G, \Delta}\,\textsc{implI1} \qquad \frac{\Gamma, F \vdash G, \Delta}{\Gamma \vdash F \Rightarrow G, \Delta}\,\textsc{implI2}$$

Finally, the annotation `!` indicates that a node must not be instantiated. In order to identify a partner formula, higher-order unification is used in general. To restrict this for instance to HO-matching but also to more specific algorithms, the keyword `check` allows the restriction to a specific algorithm.

Furthermore, without brackets around the premise/conclusion the partner is searched at top-level among the formulas in the sequent. For a bracketed premise or conclusion `[F]` the subformulas are searched as well. For instance the annotated inference `axiom` is only matched against the top-level formulas, while `axiom1: [F] ==> F` allows for the following derivation

$$\frac{\Gamma, F \Rightarrow G \vdash F, \Delta}{\Gamma, F \Rightarrow G \vdash G, \Delta}\,\textsc{axiom1}$$

The next class of annotations specifies how the formula is traversed when looking for subformulas. By default the list of all candidate subformulas is returned. Alternatively one can specify a combination of `outermost` or `innermost`, in addition with either `leftmost` or `rightmost` that the search should stop when the first is found: this can for instance be used to specify inferences to be used for a simplification using a leftmost, innermost strategy. The idea is as follows: A general term traversal can be modelled by the basic functions `visit` which performs an action on the current node (which can succeed or fail), and `traverse(n)`, which recursively invokes the traversal function on the *nth* child node of the current node. A complete term traversal is achieved by visiting all tree nodes in a certain *visiting order*. By reordering the basic functions we obtain different traversal strategies. There are also more sophisticated traversal strategies such as the level-order traversal, traversing a tree level by level, which require an additional queue to store the elements and which we therefore do not consider as a basic strategy.

Based on ideas of van den Brand [vdBKV03], we allow for different choices according to the traversal cube shown in Figure 9.3: On the first axis, we distinguish procedures that traverse the nodes of a tree bottom up and those that traverse a tree top-down. On the second axis, we classify traversal functions according to those that break the traversal on success and those that traverse the complete tree. On the third axis we distinguish methods that traverse the tree from left to right and those that traverse the tree from right to left. The backtracking behaviour is not represented in the cube.

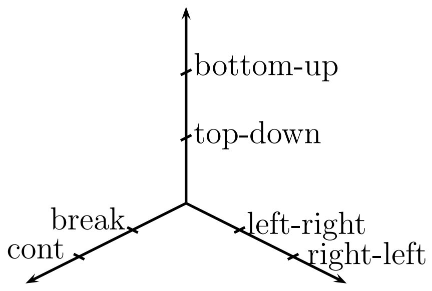


Figure 9.3: "Traversal Cube": Principal ways of traversing a tree

In general, one does not always want to compute all partners eagerly, as this might not be efficient. Therefore, we provide the keywords `LazyLeft` and `LazyRight` to specify if the formula is traversed from left to right or vice versa. In order to incrementally get *all* candidates, one can in addition specify in which order they should come: `Down` prefers outer formulas over inner formulas and `Up` the other way round. In order to not enumerate all candidates, the keywords `innermost` and `outermost` must be used to indicate the preference. In either case, the search algorithm returns a candidate subformula along with a substitution and a continuation to be used for the next candidate. That continuation is used for backtracking during the candidate search process as well as outside: if the candidate search for other required premises fails because of the used substitution, the search process uses the continuation to get the next possible candidate. If the overall inference application succeeds, the collected continuations can be saved by the strategy process and be used in case the strategy wants to backtrack.

9.3 Inference Programs

Inference application programs are LISP expressions and build the target language of the compilation process. They are built upon the underlying term data structures including basic functions such as testing for term equality, or replacing a term at a specific position by another term, which we call *atomar programs*. Based on these atomar programs, more complex programs can be built by composing atomar programs. To keep the presentation simple, we decided to give a rather high-level description of the concepts of the compilation process by showing how to decompose the overall process to small tasks which can be implemented easily. The basic ideas of the compilation process are as follows:

- Given an inference, the overall instantiation process is linearized to a sequence of instantiations.
- Each inference node is compiled to a condition, which traverses formulas of the proof state, evaluates conditions on the formulas and calls an action on success. The action is either the final modification of the proof state, or the invocation of a subsequent instantiation function.
- A traversal function for an inference node is composed of a traversal function for the sequent and a traversal function for formulas.
- An instantiation either fails, or returns a nonempty list of results including a continuation that represents parts of the search space that have not yet been considered and can be invoked to produce further results lazily.
- The objective of the compilation process is that each term is only traversed once. Therefore, proof obligations are already collected during the term traversal. Similarly, polarities of subformulas are maintained.
- Optimizations are included to minimize argument passing.

The overall construction plan of an inference is modelled by an *inference graph*, as shown in Figure 9.4 (see [Die06] for a formal definition), which models the dependencies between different inference nodes. The nodes of this graph represent all possible PAI-statuses[1], and edges between these nodes represent all possible partial argument instantiation updates, which intuitively correspond to intermediate states of the program and possible paths that can be followed.

To keep the resulting program small and efficient, we "simplify" the construction plan as follows: Given an inference, let AD denote a subset of all admissible application directions. Those nodes are marked in the graph bold and play a special role during the compilation process: The final program we want to synthesize must provide a means to reach these nodes from the root node. To obtain a minimal program satisfying these constraints, we employ a single-source shortest path algorithm to compute a minimal set of edges $\mathcal{E}$ such that all nodes in AD are reachable from the empty PAI-status using the selected edges in $\mathcal{E}$ (see [Die06] for details). Those edges that are not contained in $\mathcal{E}$ are removed.

Consider for example an inference with premises $P1, P2$ and a conclusion C. Suppose further that the set $AD = \{\langle C\rangle, \langle C, P1\rangle, \langle C, P1, P2\rangle, \langle P1, C\rangle\}$ of application directions is given, for which a possible solution is shown in Figure 9.4. The bold edges correspond to the edges that have been selected in $\mathcal{E}$. This solution is now used as follows: Compile the condition corresponding to the conclusion C, store the solutions, and process them further as follows: compile the conditions corresponding to $P2$ and execute the corresponding program on all solutions of C. Compile the condition corresponding to $P1$, invoke it on all solutions on C, store these solutions, try to further solutions by compiling the conditions corresponding to $P2$ and invoke them on the solutions $\langle P1, C\rangle$.

Of course, it is also possible to compile several inferences in parallel and to minimize the resulting program by sharing nodes of the graph. One could even go further and extend the sharing to substrings of individual formulas, which is not done in the current implementation.

[1]A PAI status is an equivalence class on partial argument instantiations given by the arguments of the inference that are instantiated

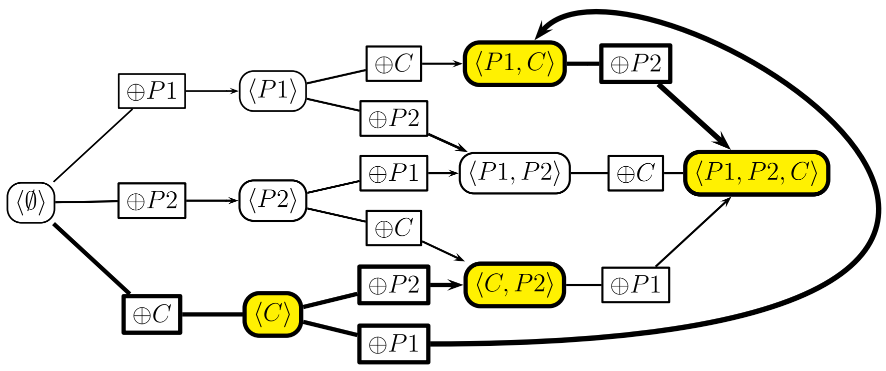


Figure 9.4: Minimal set making all marked nodes reachable

function	description	shape of resulting program
$[\![\cdot]\!]$: node → env → LISP × env	main compilation	seq → [[seq]]
$[\![\cdot]\!]_a$: node → env → LISP × env	compile action	[[seq]]
$[\![\cdot]\!]_c$: node → env → LISP × env	compile term condition	[[seq]] → form → [[seq]]
$[\![\cdot]\!]_t$: node → env → LISP × env	compile traversal	seq → (form → [[seq]]) → [[seq]]
$[\![\cdot]\!]_j$: node → env → LISP × env	compile join	[[seq]] → [[seq]] → [[seq]]

Table 9.1: Compilation functions

Table 9.1 shows the main compilation functions the overall compilation process is based upon and describes the shape of the programs that they generate. These work as follows:

- The main compilation function $[\![\cdot]\!]$ takes a node of an inference graph, as well as an environment σ, as input, and returns a LISP program that can be invoked on a sequent object and returns a list of alternatives, each corresponding to one possibility to apply the inference. Each possibility is given by a list of sequents, indicating the replacement of the input sequent by the sequents of the alternative. The environment σ is used to keep the relation between program variables and term or object variables. For example, when processing an inference node n, a program variable `n1` is generated, over which the term can be accessed, as well as `npol` and `ntaf` to access polarity and position of the node.

  Note that the resulting program can easily be lifted to the level of agendas by either traversing the open sequents of a given agenda or selecting the first sequent by default.

- The compilation function $[\![\cdot]\!]_a$ constructs a program that returns a list of possible reductions based on the parameters and positions contained in the environment, i.e., performs the induced resolution replacement rules and possibly β-decompositions to construct new sequents according to the theoretical foundations derived in Chapter 6.

- The compilation function $[\![\cdot]\!]_c$ constructs a program of the form $\lambda a.$**if** $\ldots$ **then** a **else** $\perp$ which checks whether a specific condition holds, e.g., whether a given formula is unifiable with the formula scheme of an inference node, and executes the specified action a in this case. If the condition does not succeed, a failure $\perp$ is returned.

- The compilation function $[\![\cdot]\!]_t$ constructs a traversal function of the form $\lambda a\,.\,\lambda o\,.$ traverse o a that takes an action a as input, traverses a sequent or term o according to the specification of an inference node and invokes the action a on the sub-objects. Polarities and proof obligations are automatically maintained by the traversal function.

- The compilation function $[\![\cdot]\!]_j$ joins the results of different actions together. As the binary version can easily be lifted to the n-ary case, we allow the invocation of $[\![\cdot]\!]_j$ with arbitrarily many individual results as input.

Let us now sketch the compilation process with an inference graph as input. For this purpose, we introduce the following notation: we denote a node of the inference graph containing a PAI-status by n, the argument that is added by $\arg(n)$ which is empty for the initial node, and the immediate successor nodes of n that are also PAI-status denoted by $\mathrm{succ}(n)$.

Case 1: Terminal Node

In case that $\mathrm{succ}(n) = \emptyset$, we compile the condition of $\arg(n)$, as well as the action resulting from applying the inference with the given instantiation. The resulting program is then executed by traversing the sequent and the terms according to the specification contained in the inference node.

$$\begin{aligned}
[\![n]\!]\ \sigma = \quad & \textbf{let} \\
& \quad (\mathrm{cond}, \sigma_1) = [\![\arg(n)]\!]_c\ \sigma \\
& \quad (\mathrm{trav}, \sigma_2) = [\![\arg(n)]\!]_t\ \sigma_1 \\
& \quad (\mathrm{action}, \sigma_3) = [\![n]\!]_a\ \sigma_2 \\
& \textbf{in} \\
& (\lambda\, \mathrm{seq} \centerdot \mathrm{trav}\ \mathrm{seq}\ \ \mathrm{action}, \sigma_3)
\end{aligned} \tag{9.25}$$

Case 2: Unmarked Nonterminal Node

In case that $\mathrm{succ}(n) \neq \emptyset$, there are two cases. If the node is not marked, i.e., represents an application direction that is not in the set AD, we compile the condition contained in the current node. Moreover, for each successor $s_1, \ldots, s_m$ of n we invoke the compilation function recursively (each results in a list of results plus a continuation), and concatenate the results.

$$\begin{aligned}
[\![n]\!]\ \sigma = \quad & \textbf{let} \\
& \quad (\mathrm{cond}, \sigma_1) = [\![\arg(n)]\!]_c\ \sigma \\
& \quad (\mathrm{trav}, \sigma_2) = [\![\arg(n)]\!]_t\ \sigma_1 \\
& \quad (a_1, \sigma_3) = [\![s_1]\!]\sigma_2 \\
& \qquad\qquad \vdots \\
& \quad (a_n, \sigma_{m+2}) = [\![s_m]\!]\ \sigma_{m+1} \\
& \quad (\mathrm{join}, \sigma_{m+3}) = [\![n]\!]_j\ \ \sigma_{m+2} \\
& \textbf{in} \\
& (\lambda\, \mathrm{seq} \centerdot \mathrm{trav}\, \mathrm{seq}\, (\mathrm{join}\, a_1 \ldots a_m)\, , \sigma_{m+3})
\end{aligned} \tag{9.26}$$

Case 3: Marked Nonterminal Node

In case that $\mathrm{succ}(n) \neq \emptyset$ and the node is marked, i.e., represents an application direction of the set AD, we compile the condition contained in the current node, and invoke the compilation function recursively on each successor $s_1, \dots, s_m$. In addition, we compile the action a_0 that constructs new solutions based on the current PAI-status and concatenate all results.

$$
\begin{aligned}
[\![n]\!]\ \sigma = \quad & \textbf{let} \\
& \quad (\mathrm{cond}, \sigma_1) = [\![\arg(n)]\!]_c\ \sigma \\
& \quad (\mathrm{trav}, \sigma_2) = [\![\arg(n)]\!]_t\ \sigma_1 \\
& \quad (a_0, \sigma_3) = [\![n]\!]_a\ \ \sigma_2 \\
& \quad (a_1, \sigma_4) = [\![s_1]\!]\sigma_3 \\
& \quad \quad \quad \vdots \\
& \quad (a_m, \sigma_{m+3}) = [\![s_m]\!]\ \sigma_{m+2} \\
& \quad (\mathrm{join}, \sigma_{m+4}) = [\![n]\!]_j\ \ \sigma_{m+3} \\
& \textbf{in} \\
& (\lambda\, \mathrm{seq} \,.\, \mathrm{trav}\, \mathrm{seq}\, (\mathrm{join}\, a_0\ a_1 \dots a_m)\,, \sigma_{n+4})
\end{aligned} \tag{9.27}
$$

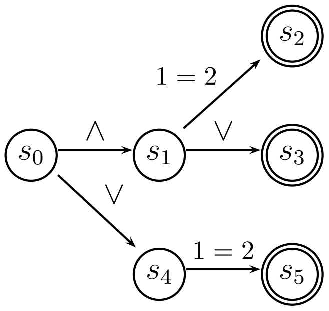
The remaining functions can either be implemented manually as library functions – in this case it is the job of the compiler to select the appropriate function and provide the arguments correctly – or they can be composed out of primitives. We have implemented the first approach, as this reduces compile time and allows the incorporation of implementation tricks that cannot be performed automatically. Let us now discuss further optimizations.

9.3.1 Explicit Matching Automata

The condition compiler $[\![\cdot]\!]_c$ generates the part of the program that checks whether a given input formula satisfies the relation indicated by the argument `check`, which is usually unification or matching. To further reduce the costs of matching, we provide a possibility to automatically construct a term-specific matching automaton, which is then compiled into the underlying programming language. The idea is to speed up the matching process by factorizing patterns, avoiding parameter passing, and to replace generic algorithms by specific instances. Such ideas have already been studied for term rewriting systems (see for example [HO82, Grä91, RR92, NWE97, Chr93]). Note that the overall program that is constructed can be seen as a matching automaton for a list of inferences. While the resulting program is reasonably efficient, further improvements can be incorporated.

We use a variant of discrimination nets (see for example [Chr93]), which are reasonably efficient and relatively easy to implement. Discrimination nets are a variant of the trie data structure. Scanning a pattern set from left to right, a tree is formed. At each point where two symbols differ a new branch is added to the tree. Instead of restricting the approach to linear patterns or using a placeholder $*$ for variables, we perform variable checks immediately. We give an example: Suppose the patterns $(A \wedge (B \vee C))$, $A \wedge A$ and $A \vee B$ are given. The abstract automaton, sharing common parts, is shown on the left of Figure 9.5. It is then easy to extract a program from this automaton, as shown on the right of Figure 9.5. Moreover, it is immediately clear how to add further constraints by strengthening the conditions at specific places, or how to add additional context information.

Let us again stress the fact that the constructed program uses only a few basic functions. Efficiency comes from sharing parts, but also from the fact that no substitutions need explicitly maintained and passed around in form of list structures. To that end, we systematically store subterms in variables (see the **let** constructs above where $|_i$ accesses the i-th subterm) and make use of these bindings during term construction.



```
lambda term
if (term-fn term = $\wedge$)
then
  let
    term1 = term$|_1$
    term2 = term$|_2$
  in
  if (term-fn term = $\vee$)
  then
    let
      term11 = term$|_1$
      term12 = term$|_2$
    in
    ⟨action-$s_2$⟩
  else
    if (term1 = term2)
    then
      ⟨action-$s_3$⟩
    end if
  else
    if (term-fn term = $\vee$)
    then
      let
        term1 = term$|_1$
        term2 = term$|_2$
      in
      if (term1 = term2)
      then
        ⟨action-$s_5$⟩
      end if
    end if
  end if
end if
```

Figure 9.5: Matching automaton and synthesized program

9.3.2 Pruning

In case deep matching is activated, the standard procedure consists of traversing the complete formula and visiting all its subformulas. Our implementation uses the following optimization rules which interrupt the traversal when specific conditions hold:

- If a previous instantiation program succeeded on a specific substructure, all subsequent instantiation programs must stop the traversal once this substructure is reached. This is due to condition (iii) of Definition 6.3.5, which prohibits the use of overlapping subformulas.
- If a previous instantiation program succeeded on a specific substructure $f|_t$ of a sequent formula f and this formula is traversed by a subsequent premise instantiation program, then at β-nodes we prune subtrees that are not on the path to $f|_t$. This is because all substructures in that tree are not α-related to the already instantiated premise (see condition (iv) of Definition 6.3.5).
- If a previous instantiation program for a conclusion succeeded on a specific substructure $f|_t$ of a sequent formula f, subsequent instantiation programs for conclusions need only to consider the same sequent formula, more specifically siblings of the instantiated conclusion. This is because of (ii) of Definition 6.3.5.
- We abort a traversal eagerly when the condition `nopob` is specified at a β-node when traversing a formula different from the conclusion.

As a simple example, consider the formula

$$\left[\left(A^+ \wedge^\beta B^+\right)^+ \Rightarrow^\beta \left(C^- \wedge^\alpha D^-\right)^-\right]^- \wedge^\alpha E^- \tag{9.28}$$

Once the conclusion has been instantiated with A^+, the subtree $(C^- \wedge^\alpha D^-)^-$ needs not to be considered, as all of its subformulas are β-related to A^+. Moreover, when instantiating an additional conclusion, the only remaining candidate is B^+.

9.3.3 Implementation Note on Traversal Functions

Traversal of formulas of a sequent or subformulas of a formula can be seen as a simple form of iteration. When performing the iteration lazily, one particular requirement is to allow the interruption of an iteration, and the resumption on a previous state of the iteration, as needed in case of backtracking. Such data structures are known as *persistent data structures* (see [DSST89]). Persistent data structures are data structures which always preserve the previous version of themselves when thay are modified.

Within our setting, two data structures are of interest, depending on whether the traversal is only used to collect further information, or whether we also want to perform a replacement of the traversed structure. The data structures are known as *backtracking iterators* (see [Fil06]) and *Huet's zipper* [Hue97], which can also be combined. The basic idea is simple: Avoid to traverse a term several times.

The idea of the zipper is to perform the traversal of a node including a subsequent replacement at a position efficiently. To that end, the zipper maintains the path from the root node of the tree to the visited node, such that the construction of the object resulting from the replacement is possible without having to traverse the term again.

The idea of backtracking iterators is to store visited nodes that have not yet been processed in a cache and use this cache in case further nodes need to be explored. The backtrack information can then be composed from the current position and the cache that has been constructed.

For a reference implementation and an evaluation of the performance we refer to [Fil06].

9.4 Discussion

Our approach combines and extends many techniques of knowledge-based proof planning, rewriting and automated reasoning (hyperresolution) to obtain a generic inference compiler that supports the generation of efficient source code. This approach has several advantages over the standard approach which encodes such heuristics in the programming language of the prover: (i) It introduces a new abstraction layer that allows the development of optimizations independently of the specification of the search strategies. (ii) Special language constructs are provided for typical problems that arise within the proof search. (iii) The compiler can perform optimizations that are too complex for a manual implementation. One disadvantage of the approach is that the compiler must touch critical parts of the actual system to achieve maximal efficiency. This can be avoided by compiling with respect to a trusted kernel – resulting in less efficient code – or by requiring a small proof checker to check the constructed proof after it has been found. Another approach is to prove the correctness of the compiler, which is a highly time consuming task. As before, the required costs can be reduced by reducing the target language to a minimum. Indeed, in the context of rewriting (see [VB98]), it has been shown how to model term traversal strategies based on a fixed point operator, sequential composition, and an operator that applies a program at the root node of a tree. In contrast, we make use of library functions that have been written once and for all, which has a positive effect on the compilation time.

9.5 Related Work

ΩMEGA's Methods and Control Rules

The effects of proof planning methods of the proof planner MULTI of the old ΩMEGA system can be specified by annotations $\oplus, \ominus$, and $\otimes$. Moreover, control rules allow the specification of preferences at choice points. Inspired by that approach, we support similar, however more elaborated and expressive features. Our language is fully embedded in the proof language and it is compiled (and optimized), resulting in efficient and optimized code. In particular, it is possible to evaluate heuristic information locally, in contrast to a global control rule interpreter that has to evaluate all control rules of a specific kind at a given choice point. Moreover, our language goes beyond that of MULTI: We allow the specification of sequent positions and backtracking behavior locally with respect to an inference. Moreover, we provide specific constructs to control the deep inference paradigm, which is not supported by MULTI.

Rewriting

The deep inference system KSg [GG04, BG07] has been implemented on top of the rewrite system MAUDE [Kah08] and TOM [KMR05], using techniques developed in the field of term rewriting. The system KSg is much closer to rewriting than our system, as a rule application consists of a single match, followed by a single replacement. In contrast to our approach, the system KSg does not feature derived rules, nor does it support the specification of search restrictions.

Several methods have been developed to efficiently compile pattern matching [War84, BM72, Aug85, Car84, FM01, Grä91, BM06, vdBHKO02, Vit96, MK98, MRV01]; moreover, strategy languages have been developed to specify different rewriting strategies (see e.g. [LV97, VB98, BKKR01]). Indeed, our approach to compile conditions to a matching automaton in an underlying programming language and to provide a language to control the search is inspired from this approach.

9.6 Summary

In this chapter we motivated several constraints on inference applications and developed a specification language to annotate inferences with user-defined search space restrictions that are required for efficient proof search (Contribution A2(i), Section 1.1). We then described how the simultaneous unification problem that arises when applying an inference can be reduced to a sequence of individual unification problems by building a matching automaton. Due to compilation techniques, our approach keeps the flexibility known from knowledge-based proof planning, but is at the same time also very efficient.

10 Reasoning at the Strategy Level: Proof Strategies

Formal proofs provide a very high degree of confidence in the correctness of a theorem, as each deduction is reduced to a sequence of primitive inference rules of a small trusted kernel. To raise the interaction level to a more abstract level, however, most interactive theorem provers provide the notion of a tactic, which is an algorithm in a meta language (usually ML) that constructs a piece of object level proof. Tactics are developed bottom up and can be put together using tacticals to obtain more complex tactics. The idea of a tactic is to reduce the number of user interactions by providing a tool that automates frequently occurring tasks and thus to obtain shorter proof scripts.

Similar to tactics, proof planning provides the notion of a method as an abstract proof operator, which originally was an annotated tactic. Methods encode heuristic knowledge in the form of "methods, procedures, and tricks of the trade, which have been used successfully by the great mathematicians over the years" as advocated by Bledsoe (see [Ble86]). They can be used within an automated setting by a proof planner (see Section 2.3 for details) and aim to avoid the combinatorial explosion of the search space as well as to provide an explanation of a proof. From a practical point of view, the main differences between methods and tactics are that (i) methods have access to and manipulate contextual information, and (ii) can postpone the verification of specific subgoals, enabling a top-down approach of proof search.

As already discussed in Section 2.2.2, both the specification of tactics and methods is difficult in state of the art proof assistants, as they require the use of the programming language of the proof assistant[1] and knowledge about its internal interfaces. Following the document-centric approach (see Section 2.2.2), it is desirable that a user can write such a tactic/method *on the fly* without leaving the document and even without leaving the proof. This would allow the non-expert user to specify proof strategies by himself, as well as to formulate highly specialized tactics to automate small parts of the proof.

In this chapter, we introduce an intermediate language which permits the user to implement new proof techniques in a compact way within the proof document. Our goal is to provide a uniform framework that captures both the bottom-up style of classical

[1]An exception is the proof assistant Coq which provides an intermediate tactic language $\mathcal{L}_{tac}$

tactics, as well as the top-down style used in proof methods. Moreover, we want the proof operators to be available in both an interactive as well as an automated setting. We call the resulting algorithms *proof strategies*.

The overall design of the language is to cover the full spectrum from declarative proof strategies – i.e., strategies which make intermediate proof states explicit but are less precise about the logical justification of the step – to procedural strategies – i.e., strategies that completely hide all internal states but focus on the application of specific proof operators in a specific order. More precisely, the key points of the language design are summarized below:

Intermediate Tactic Language: Instead of using the power of the programming language the proof assistant is implemented in, our goal consists of introducing a new *intermediate language* which is independent of the underlying programming language. The language is intended to provide a simple to use tactic language layer to bridge the gap between the predefined proof operators and the programming language of the proof assistant. In particular, the language hides the implementation details of data structures for proof states, tactics, and the concrete implementation of backtracking. The new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages and is motivated by the following reasons:

- A restricted language is easier to learn and therefore supports the user in writing his own tactics.
- Tactics can be formulated within the proof document. Therefore, they are also suited to automate small or very specific parts of a proof.
- A separate language can easily be extended and allows the change of internals and optimizations without breaking the functionality of proof search procedures.
- The semantics of the language can be described in terms of reasoning states, including the management of the search space.

We believe that declarative tactic languages offer similar advantages as declarative proof languages, namely robustness and readability, and that the trend towards declarative proof languages will carry on with declarative tactic languages. Ideally, the language is independent of both the underlying logic and the prover being used. This way, the tactic language can be connected to a prover either by providing the primitives of the tactic language within the prover or by providing a new compilation function that maps the tactic language to the primitives that are supported by the prover. An intermediate language is therefore a first step towards exchanging reasoning procedures between different proof assistants.

It is clear that such a restricted intermediate language will not be sufficient to express all possible tactics. For that reason, the language must be extensible, and we provide the possibility to inject parts of the programming language.

Support of Knowledge Filtering: Within proof assistants, it is common to organize the mathematical knowledge in modularized structured theories. Structured theories are not only designed to allow the reuse of mathematical knowledge within different contexts, but also to express a certain hierarchy on the defined concepts to organize the proof search. For example, after constructing the theory of reals, one abstracts over the concrete construction using, e.g., Cauchy sequents, and works with the concept of a real number and the properties derived for it.

To be able to take advantage of the theory structure and to make the selection of knowledge explicit, we provide a query mechanism that works on the proof context as well as on the theory. Thus, we explicitly support the cycle "select - process - search". Note that by the introduction of queries/filters, tactics become dynamic objects that depend on the context. We call such adaptive tactics *theory-aware*. This comes from the insight that restricting the number of proof operators drastically reduces the search space and allows for a more efficient solution computation, provided that the filter is not too strict (see [RS98], [MP09] for related work on relevance filtering). Moreover, it has been shown that relevance filtering is particularly important in the context of filling gaps in declarative proofs automatically (see [CKKS10]).

Local Search: Within our language, each strategy defines its own local search space including possible backtracking points. It is a particular design choice that the result of a strategy needs not to be unique, and that further solutions are explicitly maintained in the form of a continuation.

Automatic Operationalization of Knowledge: During the actual proof search process, the mathematical theory knowledge is operationalized: For example, a theorem expressing the equality of two terms can be transformed into a rewrite rule, an assertion can be translated into an inference, or a set of rewrite rules can be completed to obtain a confluent term rewriting system. Given a set of knowledge items resulting from executing a structured query, our language provides constructors to operationalize them by restricting their applicability.

Compact Proof State Patterns: Heuristic decisions are often driven by or based on a specific shape of the goal state, which can often be described by syntactic patterns. Therefore, our language comes along with a rich facility to declaratively specify conditions on terms, sequents, and proof states. Our pattern language provides notation for reasoning on subformulas, as well as dynamic patterns in the form of an ellipsis construct. Providing such patterns allows the specification of conditions in a compact way and to guide the proof search accordingly. In addition, such patterns can be used to mingle the procedural and declarative style by specifying a desired goal state as pattern (declarative) and to use a procedural algorithm to reach this state by a resource-bounded forward exploration.

First Class Support of Declarative Proofs: It is widely accepted that declarative proofs are more readable than their procedural counterparts while being more tedious to write. For example, in the ISABELLE reference manual [Pau08], it is noted that

> *properly written* ISAR *proofs become accessible to a broader audience than unstructured tactic scripts (which typically only provide operational information for the machine). Writing human-readable proof texts certainly requires some additional efforts by the writer to achieve a good presentation, both of formal and informal parts of the text. On the other hand, human-readable formal texts gain some value in their own right, independently of the mechanic proof-checking process.*

(see [Pau08] page 2). Similarly, Corbineau remarks that before the introduction of declarative proof languages, *a formal proof was merely a computer program. With this in mind, think of a person reading somebody else's formal proof, or even one of his/her own proofs but after a couple of months. Similarly to what happens with source code, this person will have a lot of trouble understanding what is going on with the proof unless he/she has a very good memory or the proof is thoroughly documented.*

(see [Cor07] page 1). Nevertheless, current state of the art proof assistants do not provide sufficient support to automate the declarative style of proof, in the sense that there is no or only little support to generate declarative proofs automatically (see Table 10.1 for an overview). Except the ISAPLANNER project (see [Dix05]), the only approaches that exist are translation based (see Section 8) and usually result in proof scripts that are too detailed, as they do not provide facilities to express and control stylistic choices[2]. More importantly, they do not allow the representation of proof search knowledge based on refinement in the style of proof planning methods.

As a result, many users still prefer the procedural style of proof – at least during proof construction – and manually rewrite it to the declarative style after a procedural proof was found. Therefore, our language explicitly supports the declarative style of proof in the form of so-called declarative tactics, which are analogously defined to procedural tactics, but whose justification is given in the form of a (partial) declarative proof script that still may need to be refined further.

	ISABELLE[3]	COQ	HOL	HOL LIGHT	ACL2	MATITA	MIZAR	PVS
proc. PL	✓	✓	✓	✓	✓	✓	×	✓
decl. PL	✓	✓	✓	✓	×	✓	✓	×
intermediate TL	×	✓	×	×	×	×	×	×
decl. PL support	✓	×	×	×	×	×	×	×
proc. → decl.	×	×	×	✓	×	✓	×	×

Table 10.1: Proof Support of the declarative style

Efficiency: While the main focus of our work is to provide mechanisms to express proof strategies in a compact and convenient format, let us point out that automation speed has an direct impact on the level of reasoning: If the proof search is slow, more control knowledge is needed to prune the search space, resulting in less general methods. Therefore, efficiency is a major concern within our implementation. We rely on compilation techniques, which result in more efficient code compared to their interpreted counterparts.

We proceed by first introducing a declarative language for the specification of procedural tactics, before we introduce declarative tactics in a second step. We conclude with a detailed discussion of related work.

[2]such as leaving out steps that are considered to be trivial.

10.1 A Declarative Language for Procedural Strategies

To get a flavor of how the proof search can be organized, we consider the following very simple example from Presburger arithmetic.

Example 10.1.1. *We consider the proof of the theorem*

$$x + 0 = x \tag{10.1}$$

in Presburger arithmetic where addition is defined recursively on the first argument. After applying induction on x, we arrive at the step case with the goal

$$x + 0 = x \vdash suc(x + 0) = suc(x) \tag{10.2}$$

Trying to apply the induction hypothesis $x + 0 = x$, we get the following alternative applications:

$$suc(x + 0) = suc(x) \rightarrow suc((x + 0) + 0) = suc(x) \tag{10.3}$$
$$suc(x + 0) = suc(x) \rightarrow suc(x + (0 + 0)) = suc(x) \tag{10.4}$$
$$suc(x + 0) = suc(x) \rightarrow suc(x + 0) + 0 = suc(x) \tag{10.5}$$
$$suc(x + 0) = suc(x) \rightarrow suc(x + 0) = suc(x) + 0 \tag{10.6}$$
$$suc(x + 0) = suc(x) \rightarrow suc(x + 0) = suc(x + 0) \tag{10.7}$$
$$suc(x + 0) = suc(x) \rightarrow suc(x) = suc(x) \tag{10.8}$$

Neglecting for a moment the well known solutions for this problem, a naive solution consists of producing all possible inference applications, selecting one to be applied and to store the others for backtracking. However, the very simple example above illustrates that this solution is too inefficient, and that further techniques to restrict the search are needed:

- Without imposing any restriction, a single inference rule might be applicable in many situations, leading to a combinatorial explosion of the search space. Having several inferences in the context, the problem gets even worse.
- Suppose that we choose the right rule application in our example, e.g., (10.7) (alternatively (10.8) also leads to the solution). This means that all the work for producing (10.3)-(10.6) was unnecessary. A more efficient approach consists of lazily unfolding the next possible step, and storing information for further unfolding the search space in case of backtracking.
- Often, a simple static analysis helps to drastically reduce the number of choice points. Reconsidering Example 10.1.1, a standard technique is to direct the equation; thus turning it into a rewrite rule. A standard idea is to allow only rewrites which make the original term "simpler", in our case to apply the equation only from left to right. Technically, this can be done by the introduction of a term ordering $>$. Directing the equation in Example 10.1.1 reduces the choice points to (10.8).
- Often, we are in the convenient situation that we know that certain backtrack points are not needed. For example, it does not matter in which order we rewrite non-overlapping subterms in the above example. When using a confluent term rewriting

system, due to meta properties, we know that backtracking will never be needed. Maintaining backtrack information in such a situation is not needed and inefficient. Even if backtracking is needed, controlling the backtracking points explicitly might avoid redundancies in the search and is therefore crucial.

All these points are directly supported by our tactic language. We give an overall overview of the language by means of several examples.

A Simple Inversion Strategy

Listing 1 shows a simple inversion strategy for propositional logic which decomposes the goal according to its structure. To that end, it tries to apply corresponding inference rules to the goal in the specified order using the **apply!** operator. The **apply!** operator applies the first operator that is applicable, and keeps the backtrack points provided by the operator. However, it does not add new backtrack points. This means that if *andi* succeeds, *impi* and *(***apply** *ore1, ore2)* are not tried. In contrast, the operator **apply** explicitly keeps backtrack points. That is, even if *ore1* succeeds, the alternative *ore2* is kept for backtracking.

```
apply!  andi, impi, (apply ore1, ore2)
```

Listing 1: A simple inversion strategy

Simplification Using Rewriting

Listing 2 shows a simplification strategy that repeatedly rewrites the proof state until no further rewrite rule can be applied. In contrast to the inversion strategy, which was static, the simplification strategy is dynamic: It explicitly contains a query that selects available equations and tries to direct them using a lexicographic path ordering. The ordering function itself is implemented as a library function in the underlying programming language and linked in the **where** clause of the query. Note that our language abstracts over implementation details, such as efficient data structures like discrimination nets [Chr93].

```
strategy simplify
  repeat
    apply (use select lhs=rhs from current
      where (greaterlpo lhs rhs) as forward
    union
    use select lhs=rhs from current
      where (greaterlpo rhs lhs) as backward)
```

Listing 2: A simplification tactic based on rewriting

Considering the strategy language in more detail we observe that the above strategy expression consists of two parts. The inner part, built by a **select** expression, selects a set of proof operators. The outer part specifies a search strategy to perform the actual search. In our example, the **select** query expresses that only the knowledge from the current theory is used. **repeat** is an iterator, which applies its argument as long as possible, but does not perform any backtracking. **backward** and **forward** are keywords to restrict the applicability of the selected knowledge to one particular direction, for equations this means either left to right or right to left.

A Simple Induction Strategy

Listing 3 shows a simple induction strategy that first applies an induction scheme and subsequently a simplification tactic to the base case and rippling to the step case. The tactic illustrates how to build complex strategies using so-called strategicals, here **thenselect**, which selects a goal and applies a strategy to it. Strategicals express how to further process proof states resulting from the first strategy application. Instead of relying on a certain order of the produced subgoals, our strategy uses a declarative matcher to select the corresponding goal. By doing so, it becomes independent of this order, which is a typical problem in standard implementations of tactic languages. Such implementations break if the induction scheme suddenly produces the step case before the base case or if it produces multiple step and base cases when only one of each was expected.

```
strategy Induction
  apply natinduct as backward
  thenselect
  cases
    * |- P 0 -> Simplification
    default -> Rippling
```

Listing 3: An induction tactic

Forward Exploration

Listing 4 shows a tactic that performs a forward exploration to derive a formula $formula$ which is a parameter[4] of the tactic. Possible goal states that shall be derived are specified in form of a declarative condition on the goal state `*,[formula]- |- *`, which evaluates to true if `formula` occurs as a negative subformula on the left-hand side of the sequent. The **solve** construct expresses that the complete search space including all alternatives must be traversed. `top(1).theorems` selects the theorems that have been proved in the current theory and all theories the theory directly imports from. To guarantee termination of the strategy, a dynamic backtrack condition is installed using the **backtrack-if** command, triggering backtracking if a certain depth is reached.

```
strategy fact
  solve
    using select * from top(1).theorems as
forward
  until *,[formula]- |- *
  backtrack-if (> depth 5)
```

Listing 4: A tactic that uses forward exploration to derive a fact given as parameter

10.1.1 Syntax

We now describe the syntax of the language in detail. The language is arranged in two levels, a query language to access the mathematical knowledge, and a strategy language which makes extensive use of these queries and annotates the result of a query with further

[4]Free variables of the tactic need to be passed as parameters.

control information to build a proof strategy. Table 10.2 summarizes the language in a BNF like notation.

Note that the language explicitly supports the use of library functions of the underlying programming language, as indicated by <function_call>. This keeps the language simple but extensible.

Organization of Knowledge in Development Graphs

The knowledge of the ΩMEGA system is organized in theories that are built on top of each other by importing knowledge from lower theories via theory morphisms. This organization is based on the notion of a *development graph* (see [AHMS02] for details).

Each theory in the development graph contains standard information like the signature of its mathematical concepts and corresponding axioms, lemmas and theorems. In addition to these notions, the development graph allows the specification of other *kinds of knowledge*, which are not necessarily affecting the semantics of a theory but which, for instance, provide valuable information for the proof procedures. An example is an ordering information for the function symbols in the signature of a theory, which can be exploited by simplification procedures. Knowledge can either manually be added to a *knowledge kind*, or automatically be classified by classification functions. For example, one either specifies an axiom to be a definition, or relies on predefined heuristics for definition detection.

Each knowledge item is attached to a specific theory and is visible in all theories that link to this theory. The links can use morphisms to transform the structures from the source theory, therefore the knowledge items are also transformed along these morphisms. For instance, a morphism that renames a function f into a function g transforms an ordering information that $f > h$ (for some function h of the signature) into the ordering information $g > h$. The default behavior for knowledge transformations is simply to transform all terms which occur inside the knowledge item.

The select Statement

The **select** statement is used to select knowledge from a specified theory, for example, the local axioms. It consists of three parts, a **selector** part, a **from** part, and a **where** part.

The From Part. The **from** part specifies the knowledge source and the theory of the development graph from which the knowledge is retrieved. The theory and the corresponding knowledge source can be accessed by their names. Moreover, we support a number of predefined keywords, e.g., to access the current theory, or the base theory. An overview of the available keywords is shown in Table 10.3.

From a global perspective, one can divide the knowledge into two parts: *direct knowledge*, and *indirect knowledge*. *Direct* knowledge is knowledge which has the form of a proof operator, i.e., an inference or a proof strategy, or can be converted to such, such as an axiom. For that purpose knowledge transformation functions need to be specified. For example the function which transforms a name to an axiom simply returns the first axiom which has the specified name. The transformation function from formulas to inferences determines exactly those inferences which were synthesized from the specified formula.

Indirect knowledge is knowledge which cannot be converted to a proof operator, but which can be used to control the search or the knowledge selection process at choice points. An example for indirect knowledge is a symbol ordering. It cannot be converted

<select>	::=	**select** <selector> **from** <source> <wherecond>?
	|	<select> **union** <select>
	|	(<select>)
<selector>	::=	*
	|	<term>
	|	(name,)* name
<source>	::=	theoryname
	|	theoryname.knowledge
<infexpr>	::=	**use** <select> <infcond>? <wherecond>?
	|	<infexpr> **union** <infexpr>
	|	<infexpr> **intersection** <infexpr>
	|	<infexpr> **difference** <infexpr>
	|	(<infexpr>)
<infcond>	::=	**as** <infdirection>
<wherecond>	::=	**where** <function_call>
<infdirection>	::=	**forward** | **backward** | **close**
<stratexpr>	::=	<infexpr>
	|	(<stratexpr>)
	|	**apply** | **apply!** <stratexpr>+
	|	**solve** <stratexpr>
	|	**repeat** <stratexpr> <untilcond>?
	|	**try** <stratexpr>
	|	<stratexpr> **then** <stratexpr>
	|	name
	|	<stratexpr> **backtrack-if** <cond>
	|	**cases** <case>+ **end**;
	|	<stratexpr> **thenselect** <stratexpr>
<case>	::=	<cond> -> <stratexpr>
<untilcond>	::=	**until** <cond>
<cond>	::=	<matcher>
	|	<function_call>
<defstrat>	::=	**strategy** name <stratexpr> **end**;
<matcher>	::=	<matchhead> <matchcond>?
<matchhead>	::=	<sequent> | var
<matchcond>	::=	**where** <function_call>
<sequent>	::=	(<termpattern>,)* <termpattern> |- <termpattern>
<termpattern>	::=	<namedterm> | [<namedterm>] <termqualifier>?
<termqualifier>	::=	+ | -
<namedterm>	::=	<term> | name:<term> | *

Table 10.2: Syntax of the procedural strategy language

all	all theories reachable from the current context
current	only the local knowledge of the current theory
base	only the underlying logic
top(n)	only the theories reachable from the current theory in n steps
"name"	only the local knowledge of the theory given by name
axioms	the axioms
theorems	already proved theorems
formulas	axioms and proved theorems
definitions	axioms which were classified as definitions
inferences	inferences, user defined and derived from axioms
local.inferences	inferences that are local to a task
strategies	user defined strategies
knowledge	stands for any knowledge

Table 10.3: Available source keywords

to a proof operator, however, it can indirectly be used to restrict the applicability of the proof operators.

Note that new knowledge kinds can easily be added to the development graph and are directly available in the query language.

The Selector Part. The selector part works on the knowledge kind specified in the **from** part and can be used to further narrow down the set of knowledge items returned by the query. In the simplest case, the selector part consists of a single `*` and all knowledge items are returned. The selector part may also consist of a set of names, in which case only those knowledge items are returned for which there is a name in the specified list of names. Finally there is the possibility to specify a term pattern. In this case only those knowledge items which correspond to a formula that matches the given term are returned. The term shares the variables with the proof context in which the query is evaluated. Free variables are interpreted as meta-variables to be instantiated by the query. These instantiated variables are passed to the **where** part and can be used there for the specification of additional constraints. We found out that it is convenient for the matching to remove all leading quantifiers of formulas corresponding to knowledge items.

The Where Part. The **where** part can be used to specify additional constraints the knowledge items of the query have to satisfy. A variable binding which stems from a pattern matching in the selector part is available for evaluation of the expression in the **where** part. Listing 5 shows an example of a **select** expression consisting of a selector part, a **from** part, and a **where** part. The query returns those axioms from the current theory which are equations (after removing the leading quantifiers) $lhs = rhs$ and binds the left-hand side of the equation to the variable lhs, and the right-hand side of the equation to the variable rhs. In the **where** part, it is checked whether lhs is greater than rhs using the LPO with the symbol ordering stored in the development graph as measure for the terms.

```
select lhs=rhs from current.axioms where (greaterlpo
lhs rhs)
```

Listing 5: Example of a **select** expression which binds variables lhs and rhs for the specification of additional constraints

The use Statement

The **use** statement can be invoked directly after a select statement and performs two tasks: (1) Knowledge items are transformed to proof operators using the installed transformation functions. If no conversion is possible, the user is informed that the query cannot be correctly interpreted. (2) The obtained proof operators are augmented by further control information. For inferences there is the possibility to restrict their application direction using the **as** keyword. **Backward** restricts the applicability of inferences to those where all conclusions are instantiated, **forward** to those where no conclusions are instantiated, and **close** to those where all premises and conclusions are instantiated.

Moreover, the automation API can be accessed via the underlying programming language inside a **where** condition. In contrast to the situation in the **select** expression, the **where** condition in a **use** expression is evaluated at runtime after the instantiation of the proof operator for a concrete task. This allows for example the proof strategy to check whether the instances of a rewrite rule can be directed if the rule itself cannot, as is the case for permutative rewrite rules[5]. Note that a **where** condition can be attached to both the **select** expression and the **use** expression, and thus allows for ambiguous formulations as shown in Listing 6:

```
use select ... from ... where c
```

Listing 6: Ambiguous use of a **where**

Such expressions are internally disambiguated by operator precedence which assigns the **where** construct to the closest expression, that is, the **select**. If the user wants to attach a **where** condition to the use expression, the **select** statement must be enclosed in brackets.

Several **use** statements can be combined by means of standard list operators. Currently, we support concatenation of the results of two **use** constructs with the **union** operator, an **intersection** operator which returns only those inferences which are returned by both **use** expressions, and a **difference** operator, which returns only those inferences which are contained in the first but not in the second result list. Note that equality of inferences includes equality of their annotations.

Strategy Constructors

The final step in the specification of a proof strategy consists of augmenting the specified list of proof operators by control information needed for their execution. Essentially, this consists of a specification of a condition for success and termination. Moreover, there is the possibility to specify a condition for failure, i.e., to directly invoke backtracking. An overview of the strategy constructors is shown in Table 10.4. These constructors are classified into the three categories **selector**, **iterator** and **combinator**.

Selectors. A list of given proof operators augmented by control information can be instantiated, resulting in a list of **PAIs** satisfying a specified set of constraints. **Selector** expressions determine how the **PAIs** are produced, and which **PAI** to choose among the set of produced **PAIs**. For example, the **apply** selector starts to instantiate the proof operators and applies the first which is applicable and satisfies the specified constraints. Another constructor is the **cases** constructor. It consists of a list of condition action pairs,

[5]An equation is a permutative rewrite rule if the left-hand side and right-hand side are the same up to renaming of variables.

apply	selector	applies the first proof operator that succeeds, keeps remaining proof operators for backtracking.
apply!	selector	applies the first proof operator that succeeds, does not store backtracking information.
solve	iterator	applies the strategy until it fails or a solution is found.
repeat	iterator	applies the strategy until it is not applicable anymore or until the condition evaluates to true.
try	combinator	applies the strategy and doesn't fail if the strategy fails.
then	combinator	applies the first strategy and then the second strategy. Fails if the first strategy fails. If the second strategy fails then the first strategy backtracks internally and the second strategy is invoked again.
backtrack-if	combinator	adds a backtrack event specified in condition to the strategy.
cases	selector	inspects a task and triggers a strategy application if the condition is met.
thenselect	combinator	executes a strategy and applies a second strategy to each of the resulting tasks

Table 10.4: Strategy constructors with corresponding classification

where the condition is encoded in the form of a matcher or a function call. It executes the first action whose condition is satisfied. Similar to the matching in **select**-expressions bound variables are passed to subsequent expressions.

```
cases
  * |- ~x -> apply use select Contradiction from
base.inferences
  default -> apply use select same from
base.inferences;
```

Listing 7: The **cases** construct

Listing 7 shows a simple example using the **cases** selector. The matcher $*|- \sim x$ encodes the condition that the goal of the current task is a negated formula. In this case, proof by contradiction is performed. As default case the inference "same", which essentially tries to simplify the goal by a subformula of an assumption which unifies with a subformula of a goal, is applied. To allow the formulation of shorter strategy expressions, several shorthand notations are implemented. For example, **use** and **select** can be omitted in situations where the context is clear, resulting in the simplified strategy shown in Listing 8.

```
cases
  * |- ~x -> apply Contradiction
  default -> apply same;
```

Listing 8: Shorthand notation

Iterators. An **iterator** encodes a loop together with a default backtracking behavior. So far, we support two iterators, **solve** and **repeat**. **Solve** tries to close the task to which it was applied to by repeatedly applying the specified proof operators. If none of them is applicable, it fails or backtracks. **Repeat** applies the specified proof operators until none of them is applicable anymore. Additionally, a termination condition can be specified using the **until** keyword.

Combinators. Combinators combine strategy expressions. We support four strategy combinators, **try**, **then**, **backtrack-if**, and **thenselect**. Whereas the first two have their standard meaning, the latter need further explanation. **Backtrack-if** adds a failure condition to an *event store* and thus explicitly invokes backtracking. **Thenselect** splits the tasks resulting from a strategy application and applies another strategy to each of them, as already shown in Listing 3.

Matcher. Matcher are either defined on terms, sequents, or proof states. In addition to match top-level formulas, we also allow the matching of subformulas. This is in particular useful because we allow the application of inferences deeply inside formulas. We use the notation `[term]` to say that `term` can be a subterm. Moreover, we support the restriction of subformulas to specific polarities (see [Wal90]), where we use `+` to indicate a subformula with positive polarity (intuitively a goal to be shown) and `-` to indicate a subformula with negative polarity (intuitively a hypothesis).

10.1.2 Semantics of the Query Language

To illustrate the semantics of our query language we consider the evaluation of the following **use** expression:

$$\textbf{use}\ (\textbf{select}\ \mathit{sel}\ \textbf{from}\ \mathit{source}\ \textbf{where}\ \mathit{condition})\ \textbf{where}\ \mathit{condition} \tag{10.9}$$

Our evaluation algorithm works on a list of so-called *configurations* and processes the query in four steps. A configuration is a pair $\langle \mathcal{E} \parallel i \rangle$ consisting of an environment $\mathcal{E}$ and an object i, which is possibly a list. In this case we write $[i_1, \ldots, i_m]$ to denote a list of items $i_1, \ldots, i_m$; $[]$ denotes the empty list. The environment $\mathcal{E}$ maintains information needed to evaluate the query, such as the name of the current theory or variable bindings. Given an environment $\mathcal{E}$ and a variable x, we write $\mathcal{E}(x)$ for the value of the variable x, which is $\perp$ if the variable is unbound. Moreover, we write $\mathcal{E}, x \mapsto i$ to denote the change from environment $\mathcal{E}$ to the environment $\mathcal{E}'$ in which x gets assigned the value i. For a substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we write $\mathcal{E}, \sigma$ to indicate the environment $\mathcal{E}, x_1 \mapsto t_1, \ldots, x_n \mapsto t_n$, where already existing bindings are updated.

Having introduced the notion of a configuration, we can now describe the evaluation of expression (10.9), which proceeds as follows:

First, the **from** part is evaluated, resulting in a list of all knowledge items found in the source. The **select** part can then filter the resulting list by pattern matching and provides a variable binding for the evaluation of the **where** condition. Finally the knowledge items are converted to proof operators.

The semantics of the query language is shown in Figure 10.1. We use $\cup$ to denote the append operator for lists, $\mathit{prep\_match}(i)$ to prepare an item for the matching process, and $\mathit{match}(\mathit{pattern}, \mathit{term})$ to invoke the matching process. We will explain the semantics by means of an example.

Consider the theory of Presburger arithmetic shown in Figure 10.2, where **type** *nat = 0 | suc(nat)* defines a new inductive type for natural numbers. As running example we consider the evaluation of the statement

$$\textbf{use select}\ \mathit{lhs} = \mathit{rhs}\ \textbf{from}\ \mathit{current.formulas}\ \textbf{where}\ (\text{greaterlpo}\ \mathit{lhs}\ \mathit{rhs})\ \textbf{as forward} \tag{10.10}$$

$$\frac{\langle \mathcal{E} \parallel [] \rangle : \textbf{from } \mathit{src} \to \langle \mathcal{E} \parallel [i_1, \ldots, i_n] \rangle}{\langle \mathcal{E} \parallel [] \rangle : \textbf{select } \mathit{sel} \textbf{ from } \mathit{src} \textbf{ where } \mathit{cond} \to \textbf{select } \mathit{sel}\ \langle \mathcal{E} \parallel [i_1, \ldots, i_n] \rangle \textbf{ where } \mathit{cond}}$$

$$\frac{\langle \mathcal{E}, \mathit{item} \mapsto i_1 \parallel i_1 \rangle : \textbf{where } \mathit{cond} \to \mathit{res}_1 \ \ldots\ \langle \mathcal{E}, \mathit{item} \mapsto i_n \parallel i_n \rangle : \textbf{where } \mathit{cond} \to \mathit{res}_n}{\langle \mathcal{E} \parallel [] \rangle : \textbf{select } * \ \langle \mathcal{E} \parallel [i_1, \ldots, i_n] \rangle \textbf{ where } \mathit{cond} \to \cup_{i=1}^{n} \mathit{res}_i}$$

$$\frac{\begin{array}{c}\langle \mathcal{E}, \mathit{item} \mapsto i_1 \parallel i_1 \rangle : \text{MATCH } \mathit{pattern} \textbf{ where } \mathit{cond} \to \mathit{res}_1 \\ \vdots \\ \langle \mathcal{E}, \mathit{item} \mapsto i_n \parallel i_n \rangle : \text{MATCH } \mathit{pattern} \textbf{ where } \mathit{cond} \to \mathit{res}_n\end{array}}{\langle \mathcal{E} \parallel [] \rangle : \textbf{select } \mathit{pattern}\ \langle \mathcal{E} \parallel [i_1, \ldots, i_n] \rangle \textbf{ where } \mathit{cond} \to \cup_{i=1}^{n} \mathit{res}_i}$$

$$\frac{\langle \mathcal{E} \parallel i \rangle : \mathit{prep\_match}(i) \to \mathit{term} \qquad \langle \mathcal{E} \parallel i \rangle : \mathit{match}(\mathit{pattern}, \mathit{term}) \to \sigma}{\langle \mathcal{E} \parallel i \rangle : \text{MATCH } \mathit{pattern} \textbf{ where } \mathit{cond} \to \langle \mathcal{E} \cup \sigma \parallel i \rangle \textbf{where } \mathit{cond}}$$

$$\frac{\langle \mathcal{E} \parallel i \rangle : \mathit{prep\_match}(i) \to \mathit{term} \qquad \langle \mathcal{E} \parallel i \rangle : \mathit{match}(\mathit{pattern}, \mathit{term}) \to \bot}{\langle \mathcal{E} \parallel i \rangle : \text{MATCH } \mathit{pattern} \textbf{ where } \mathit{cond} \to []}$$

$$\frac{}{\langle \mathcal{E} \parallel i \rangle : \textbf{where } \mathit{cond} \to [i]}\ \text{IF } \mathcal{E} \models \mathit{cond} \qquad \frac{}{\langle \mathcal{E} \parallel i \rangle : \textbf{where } \mathit{cond} \to []}\ \text{IF } \mathcal{E} \not\models \mathit{cond}$$

Figure 10.1: Semantics for **select** expressions

```
type nat = 0 | suc(nat)
definition +:nat→nat→ nat, infixl
  ∀x.  0 + x = x
  ∀x,y.suc(x) + y = suc(x + y)
```

Figure 10.2: Theory Presburger

within the theory "Presburge_Axioms". Evaluating the query within the theory "Presburger_Axioms" and a given environment $\mathcal{E}$ means to evaluate the query with respect to the following, so-called *initial configuration*:

$$\langle \mathcal{E}, \mathit{current} \mapsto \mathit{Presburger\_Axioms} \parallel [] \rangle \tag{10.11}$$

Step 1: Evaluation of the from part. As the query is evaluated in the context of the theory "Presburger_Axioms", *current* is bound to *Presburger_Axioms.* According to our semantics, the **from** part (first rule) evaluates to a configuration consisting of an environment $\mathcal{E}$ and a list of items $[i_1, \ldots, i_n]$. In our case, the items are exactly all formulas of the theory "Presburger_Axioms" and also contain the axioms for the data type. Thus we obtain the following configuration[6]:

[6]Note that axioms for induction, injectivity, and case distinctions are generated automatically by declaring an inductive data type

$$\begin{array}{ll}\langle \mathcal{E}, \mathit{current} \mapsto \mathit{Presburger\_Axioms} \parallel [& \forall x.0+x=x, \forall x,y.suc(x)+y=suc(x+y), \\ & \forall P.\ P(0) \wedge (\forall x.P(x) \Rightarrow P(suc(x))) \Rightarrow \forall x.P(x), \\ & \forall x,y.\ suc(x)=suc(y) \Leftrightarrow x=y, \\ & \forall x.\ 0 \neq suc(x)]\rangle \end{array} \tag{10.12}$$

Step 2: Evaluation of the select part. In the second step, the **select** part, which is either $*$ or a term pattern within the term language of the theory, is evaluated with respect to the result from the previous step. For each variant there is a corresponding rule in Figure 10.1.

In the first case, the use of $*$, each of the items returned by the **select** expression is passed to the evaluation of the **where** part, where for each item the environment is enriched by the built-in variable *item* which is bound to the current item. This way the item can be accessed in the **where** part. That is, the expression "**where** cond" is actually "**where** λ item. cond".

In the latter case, where a term in the language of the theory is used to specify a pattern, all returned items have to match the pattern given by the term. Moreover, the bindings obtained by the matching are added to the environment. As in the previous case, each item is separately processed, and the individual results combined using the append operator $\cup$. In contrast to the previous rule, only those items satisfying the matching condition are passed for further evaluation. The matching process itself is encoded in the basic function *match*, which either fails or returns a substitution σ. In the latter, case the variable binding of the substitution σ is added to the environment $\mathcal{E}$, such that the binding is available for processing in the **where** part. In order for an item to be processable by *match*, the item is preprocessed by *prep_match*.

In our running example, the **select** part is evaluated with respect to the data (10.12). As the selector is a pattern $lhs = rhs$, the data is preprocessed, resulting in

$$\begin{array}{ll} [\langle \mathit{current} \mapsto \mathit{Presburger\_Axioms}, \mathit{item} & \mapsto 0+x=x \parallel \forall x.0+x=x\rangle \\ \langle \mathit{current} \mapsto \mathit{Presburger\_Axioms}, \mathit{item} & \mapsto suc(x)+y=suc(x+y) \parallel \\ & \qquad \forall x,y.suc(x)+y=suc(x+y)\rangle \\ \langle \mathit{current} \mapsto \mathit{Presburger\_Axioms}, \mathit{item} & \mapsto P(0) \wedge (\forall x.P(x) \Rightarrow P(suc(x))) \Rightarrow \forall x.P(x) \parallel \\ & \forall P.P(0) \wedge (\forall x.P(x) \Rightarrow P(suc(x))) \Rightarrow \forall x.P(x)\rangle \\ \langle \mathit{current} \mapsto \mathit{Presburger\_Axioms}, \mathit{item} & \mapsto suc(x)=suc(y) \Leftrightarrow x=y \parallel \\ & \qquad \forall x,y.suc(x)=suc(y) \Leftrightarrow x=y\rangle \\ \langle \mathit{current} \mapsto \mathit{Presburger\_Axioms}, \mathit{item} & \mapsto 0 \neq suc(x) \parallel \forall x.0 \neq suc(x)\rangle] \end{array} \tag{10.13}$$

For each configuration its item is matched against the term $lhs = rhs$. For the first two items the matching succeeds with the matcher

$$\sigma_1 = \{lhs \mapsto 0+x, rhs \mapsto x\} \tag{10.14}$$

and

$$\sigma_2 = \{lhs \mapsto suc(x)+y, rhs \mapsto suc(x+y)\}, \tag{10.15}$$

respectively. For the other items, matching fails. We thus obtain the following result:

$$\begin{aligned}[\langle current \mapsto Presburger\_Axioms, \quad & lhs \mapsto 0+x, rhs \mapsto x, item \mapsto 0+x=x \parallel \\ & \forall x.0+x=x\rangle \\ \langle current \mapsto Presburger\_Axioms, \quad & lhs \mapsto suc(x)+y, rhs \mapsto suc(x+y), \\ & item \mapsto suc(x)+y = suc(x+y) \parallel \\ & \forall x,y.suc(x)+y = suc(x+y)\rangle]\end{aligned} \tag{10.16}$$

Step 3: Evaluation of the where part. Finally, the condition encoded in the **where** part of the query is evaluated with respect of the updated environment $\mathcal{E}$. There are two rules: If the condition is satisfied in the environment $\mathcal{E}$, a singleton list containing the given item is returned. We use lists to allow for a simple collection process (appending using the $\cup$ operator). If the condition is not satisfied in $\mathcal{E}$, the empty list is returned. Note that the **where** part can be left out. This case is internally handled by adding the condition *true* which always succeeds.

In our example, the condition checks whether the term bound to the variable *lhs* is greater than the term bound to the variable *rhs* with respect to the lexicographic path ordering maintained by the theory, which is based on the symbol ordering induced by the definitions. As the condition is satisfied for all items, we get the following list as overall evaluation result of the **select** statement:

$$[\forall x.0+x=x, \forall x,y.suc(x)+y = suc(x+y)] \tag{10.17}$$

Step 4: Evaluation of the use part. Finally, the formulas returned by (10.17) are transformed to inferences by means of the **use** construct. In our example, the **use** construct is enriched by the annotation **as forward**, meaning that the resulting inferences are only allowed to be applied in this particular direction. As there is no **where** construct attached to the **use** construct, no conditions need to be added to the inferences. Thus, we obtain the following rewrite rules as final result:

$$0+x \to x \tag{10.18}$$

$$suc(x)+y \to suc(x+y) \tag{10.19}$$

10.1.3 Semantics of Strategy Constructors

In this section, we give a semantics for the evaluation of our language constructs, based on the notion of an agenda and an inference (see Chapter 6). In order to come close to an implementation, we give a deterministic algorithm which explicitly keeps track of possible choices which have not yet been considered. Instead of explicitly producing all choices at each step, it is more efficient to implicitly maintain these choices in the form of a *continuation*, which is most easily understood as a program producing the next choice on demand. If executed, a continuation either fails, or returns a new choice, i.e., a new state, together with a new continuation.

During the evaluation of combined expressions, it will be necessary to combine continuations of its subexpressions to get a continuation of the overall expression. We will see that we can also use our language constructs for that. Therefore, we define our evaluation mechanism to work not only for strategies, but more generally for arbitrary programs. Indeed, a strategy is a special program.

We now present the semantics of the evaluation mechanism, which works on configurations

$$\langle A \parallel \mathcal{C} \parallel p\rangle \tag{10.20}$$

consisting of an agenda A, an event store $\mathcal{C}$, containing dynamic backtrack conditions, and a program p. As introduced in Chapter 6, an agenda is a set of tasks, where each task represents a subproblem to be solved. Thus, $\emptyset$ represents an agenda where all subproblems have been solved. We denote with ϵ a fully executed program as well as the empty event store. For a better understanding of the evaluation, it is a good idea to focus on the first and third component of the configuration first, as these are modified in each step. The second component, the event store $\mathcal{C}$, is only modified by the **backtrack − if** command.

Given a strategy expression *exp* and an agenda A, the *initial configuration* is given by

$$\langle A \parallel \epsilon \parallel exp\rangle \tag{10.21}$$

Intuitively, the goal of the evaluation algorithm is to execute the program *exp* and thereby producing a proof object for the problem contained in the agenda A. This is obtained by executing *exp*, which results either in a failure, indicated by $\perp$, or in a pair

$$\langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c \tag{10.22}$$

Intuitively, c contains the part of the search space which has not yet been traversed and can be used to produce alternative proofs for the problem represented by agenda A.

There are two programs which can directly be executed, namely an inference and a continuation. Following the ideas of Elan [BKKR01], we call both *primal strategies*. Instead of directly returning the result of the execution, we additionally check whether the solution triggers a backtrack event of the event store $\mathcal{C}$. If so, the result is discarded and the continuation is recursively evaluated in order to get a new configuration which does not trigger the backtrack event. If this is not possible or the program failed in the first place, the failure is returned. Thus we get the following three rules:

Primal Strategies:

$$\frac{}{\langle A \parallel \mathcal{C} \parallel p\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c}\ \textsc{If}\ \forall \phi \in \mathcal{C}.A' \not\models \phi$$

$$\frac{}{\langle A \parallel \mathcal{C} \parallel p\rangle \to \perp}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c \qquad \langle \star \parallel \star \parallel c\rangle \to res}{\langle A \parallel \mathcal{C} \parallel p\rangle \to res}\ \textsc{If}\ \exists \phi \in \mathcal{C}.A' \models \phi$$

In the rule above, $\langle \star \parallel \star \parallel c\rangle$ emphasizes that the execution of c is independent of the current context, as c restores the context by itself.

To get an intuitive understanding of the concept and the functioning of continuations, consider the inference I, representing the rewrite rule $X + 0 \to X$[7], and the agenda A containing only a single task $\{\vdash suc(x+0) = suc(x)+0\}$. Its initial configuration is given by

$$\langle \{\vdash suc(x + 0) = suc(x) + 0\} \parallel \mathcal{C} \parallel I\rangle \tag{10.23}$$

[7]we use an upper case X to emphasize the fact that X is a meta-variable that can be instantiated

Note that there are two possibilities to apply the rewrite rule $X+0 \to X$ by binding X to x, respectively to $suc(x)$. Rather than producing one successor agenda for each possible match, the execution of I results in the pair

$$\langle \{\vdash suc(x) = suc(x) + 0\} \parallel \mathcal{C} \parallel \epsilon\rangle : c \tag{10.24}$$

where c contains the information of how to further evaluate I, implicitly keeping track of all remaining alternatives. In our example, the evaluation of c in any context thus yields

$$\langle \{\vdash \{suc(x) + 0 = suc(x)\} \parallel \mathcal{C} \parallel \epsilon\rangle : c' \tag{10.25}$$

where c' provides no further results, i.e., $\langle A \parallel \mathcal{C} \parallel c'\rangle \to \bot$ for any context $A, \mathcal{C}$.

Note that any configuration $\langle A \parallel \mathcal{C} \parallel p\rangle$, where p is a strategy or a continuation, is also a continuation as it evaluates to a new configuration-continuation pair.

Backtrack If:
The **backtrack − if** combinator installs a backtrack condition in the event store $\mathcal{C}$ and leaves the other components of the configuration unchanged. This condition is used in primal strategy expressions to check whether the result has a desired form.

$$\frac{\langle A \parallel \mathcal{C} \cup \{\phi\} \parallel s\rangle \to \bot}{\langle A \parallel \mathcal{C} \parallel \textbf{backtrack} - \textbf{if}\ \phi\ s\rangle \to \bot} \qquad \frac{\langle A \parallel \mathcal{C} \cup \{\phi\} \parallel s\rangle \to \langle A' \parallel \mathcal{C} \cup \{\phi\} \parallel p\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{backtrack} - \textbf{if}\ \phi\ s\rangle \to \langle A' \parallel \mathcal{C} \parallel p\rangle : c}$$

Apply:
As **apply** operates on lists of arguments, we inductively define its evaluation. In the base case **apply** gets a single program p (think of a strategy s) as input. The program is executed and the result returned.

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \to res}{\langle A \parallel \mathcal{C} \parallel \textbf{apply}\ p\rangle \to res}$$

Note that res is either a configuration continuation pair or the symbol $\bot$ representing failure.

In the step case, **apply** evaluates its first argument. If the execution succeeds, i.e., results in a new configuration $\langle A' \parallel \mathcal{C} \parallel \epsilon\rangle$ and a continuation c, the new configuration is returned and a new continuation c' built by extending c by **apply** $p_2\ \dots\ p_n$, performed by the operator @. If the execution fails, the operator recursively calls itself with the remaining arguments. We first give the rules of the @ operator before continuing with **apply**.

$$\frac{\langle \star \parallel \star \parallel c_1\rangle \to \bot \qquad \langle \star \parallel \star \parallel c_2\rangle \to res}{\langle \star \parallel \star \parallel c_1@c_2\rangle \to res} \qquad \frac{\langle \star \parallel \star \parallel c_1\rangle \to \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c_1'}{\langle \star \parallel \star \parallel c_1@c_2\rangle \to \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c_1'@c_2}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \langle A' \parallel \mathcal{C} \parallel p_1\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{apply}\ p_1\ \dots\ p_n\rangle \to \langle A' \parallel \mathcal{C} \parallel p_1\rangle : c@\langle A \parallel \mathcal{C} \parallel (\textbf{apply}\ p_2\ \dots\ p_n)\rangle}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \bot \qquad \langle A \parallel \mathcal{C} \parallel \textbf{apply}\ p_2\ \dots\ p_n\rangle \to res}{\langle A \parallel \mathcal{C} \parallel \textbf{apply}\ p_1\ \dots\ p_n\rangle \to res}$$

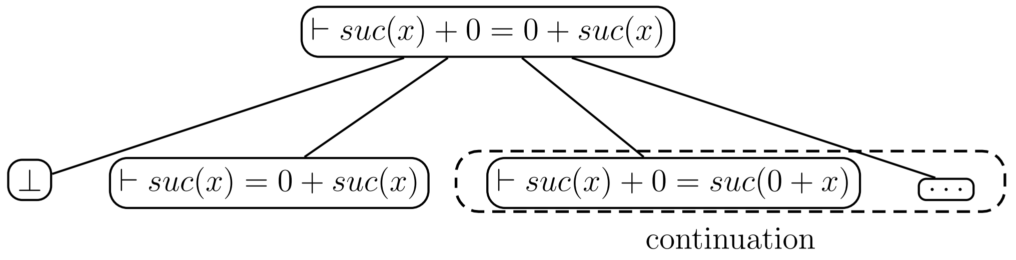
As before, we give a simple example to illustrate the concatenation of continuations. Consider the evaluation of the configuration

$$\langle \{\vdash suc(x) + 0 = 0 + suc(x)\} \parallel \epsilon \parallel \textbf{apply}\ I_1\ I_2\ I_3\ \ldots\ I_n\rangle \tag{10.26}$$

where I_1, I_2, I_3 represent the rewrite rules $x + 0 \to 0 + x$ (x constant), $X + 0 \to X, X + suc(Y) \to suc(X + Y)$, respectively. First, **apply** tries to apply I_1, which fails. I_2 can be applied, resulting in

$$\langle \{\vdash suc(x) + 0 = suc(0 + x)\} \parallel \epsilon \parallel \epsilon\rangle : c \tag{10.27}$$

where c stores all possibilities to apply $I_3, \ldots, I_n$ to the task. We can graphically represent this as follows:



The variant **apply!** is similar to apply, but does not keep backtrack points:

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \to res}{\langle A \parallel \mathcal{C} \parallel \textbf{apply!}\ p\rangle \to res} \qquad \frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \langle A' \parallel \mathcal{C} \parallel p_1\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{apply!}\ p_1\ \ldots\ p_n\rangle \to \langle A' \parallel \mathcal{C} \parallel p_1\rangle : c}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \bot \qquad \langle A \parallel \mathcal{C} \parallel \textbf{apply!}\ p_2\ \ldots\ p_n\rangle \to res}{\langle A \parallel \mathcal{C} \parallel \textbf{apply!}\ p_1\ \ldots\ p_n\rangle \to res}$$

Try:
Given an agenda, the expression **try** s is evaluated by first evaluating the strategy s. If the execution of s fails, the original agenda is returned, otherwise the result of the evaluation.

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \to \bot}{\langle A \parallel \mathcal{C} \parallel \textbf{try}\ p\rangle \to \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : \bot} \qquad \frac{\langle A \parallel \mathcal{C} \parallel p\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{try}\ p\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c}$$

Then:
The expression p_1 **then** p_2 first evaluates p_1. If p_1 fails, then p_1 **then** p_2 fails. Otherwise p_2 is executed on the result of p_1 and the new continuation is built. Failure of p_2 on the result of p_1 triggers the computation of a new result in p_1.

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \bot}{\langle A \parallel \mathcal{C} \parallel p_1\ \textbf{then}\ p_2\rangle \to \bot}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c_1 \qquad \langle A' \parallel \mathcal{C} \parallel p_2\rangle \to \langle A'' \parallel \mathcal{C} \parallel \epsilon\rangle : c_2}{\langle A \parallel \mathcal{C} \parallel p_1\ \textbf{then}\ p_2\rangle \to \langle A'' \parallel \mathcal{C} \parallel \epsilon\rangle : \langle A \parallel \mathcal{C} \parallel c_1\ \textbf{then}\ p_2\rangle @ c_2}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p_1\rangle \to \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c \qquad \langle A' \parallel \mathcal{C} \parallel p_2\rangle \to \bot \qquad \langle \star \parallel \star \parallel c\ \textbf{then}\ p_2\rangle \to res}{\langle A \parallel \mathcal{C} \parallel p_1\ \textbf{then}\ p_2\rangle \to res}$$

As illustrating example let I_1, I_2, I_3 represent the inferences from above and let the following expression be given:

$$\langle\{\underline{T_0}\} \parallel \epsilon \parallel (\textbf{apply } I_2\ I_1) \textbf{ then } (\textbf{apply } I_2)\rangle \tag{10.28}$$

where

$$T_0 : suc(0+x) = suc(x)+0 \qquad T_1 : suc(0+x) = suc(x) \tag{10.29}$$

$$T_2 : = suc(x+0) = suc(x)+0 \qquad T_3 : suc(x) = suc(x)+0 \tag{10.30}$$

First, I_2 is evaluated, resulting in the task T_1 and the continuation to apply **apply** I_1 on T_0. On T_1, I_2 is evaluated, which fails. Consequently, the continuation is unfolded, and I_1 evaluated on T_0, resulting in T_2 and the empty continuation. **apply** I_2 is called again, resulting in T_3, and the empty continuation.

Solve:
The expression **solve** s applies s repeatedly to an agenda A until all tasks of A have been solved, or all possibilities have been tried out without success, in which case a failure is returned.

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle \emptyset \parallel \mathcal{C} \parallel \epsilon\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow \langle \emptyset \parallel \mathcal{C} \parallel \epsilon\rangle : c} \qquad \frac{\langle A \parallel \mathcal{C} \parallel p\rangle \rightarrow \bot}{\langle A \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow \bot}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c_1 \qquad \langle A' \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow \langle \emptyset \parallel \mathcal{C} \parallel \epsilon\rangle : c_2}{\langle A \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow \langle \emptyset \parallel \mathcal{C} \parallel \epsilon\rangle : (c_2 @ c_1)}$$

$$\frac{\langle A \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c_1\ \langle A' \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow \bot\ \langle \star \parallel \star \parallel c_1 \textbf{ then solve}\, p\rangle \rightarrow res}{\langle A \parallel \mathcal{C} \parallel \textbf{solve}\, p\rangle \rightarrow res}$$

As an example consider the evaluation of the configuration

$$\langle \vdash suc(x)+0 = 0+suc(x) \parallel \epsilon \parallel \textbf{solve } (\textbf{apply } I_1\ I_2\ I_3)\rangle \tag{10.31}$$

which essentially works as follows:

$$\langle\{\vdash suc(x)+0 = 0+suc(x)\} \parallel \epsilon \parallel \textbf{solve } (\textbf{apply } I_1\ I_2\ I_3)\rangle \tag{10.32}$$

$$\rightarrow\langle\{\vdash suc(x) = 0+suc(x)\} \parallel \epsilon \parallel \textbf{solve } (\textbf{apply } I_1\ I_2\ I_3)\rangle : c_1 \tag{10.33}$$

$$\rightarrow\langle\{\vdash suc(x) = suc(0+x)\} \parallel \epsilon \parallel \textbf{solve } (\textbf{apply } I_1\ I_2\ I_3)\rangle : c_2 \tag{10.34}$$

$$\rightarrow\langle\{\vdash suc(x) = suc(x+0)\} \parallel \epsilon \parallel \textbf{solve } (\textbf{apply } I_1\ I_2\ I_3)\rangle : c_3 \tag{10.35}$$

$$\rightarrow\langle \emptyset \parallel \epsilon \parallel \epsilon\rangle : c_4 \tag{10.36}$$

$$\rightarrow\langle \emptyset \parallel \epsilon \parallel \epsilon\rangle : c_4 @ c_3 @ c_2 @ c_1 \tag{10.37}$$

The first three steps recursively trigger the evaluation of a **solve** expression, finally, all continuations are step by step combined.

Repeat:
The **repeat** construct is similar to the **solve** construct with the difference that the program p is applied to each task of the initial agenda until it cannot be applied any further or the task meets a specified condition, which can be given in the **until** construct. We

will subsume the case where the **until** is not explicitly given by implicitly assuming the condition $\phi = false$. For a better understanding it is useful to think of the **repeat** construct to perform a simplification. Note that for efficiency the operator does not support backtracking. As a consequence the **backtrack – if** operator does not affect directly the evaluation of the **repeat** construct but only the strategies on a deeper level.

$$\frac{}{\langle \{\} \parallel \mathcal{C} \parallel \textbf{repeat}\, p\ \textbf{until}\ \ \phi\rangle \rightarrow \langle \{\} \parallel \mathcal{C} \parallel \epsilon\rangle : \bot}$$

$$\frac{\langle \{\underline{T_1}\} \parallel \mathcal{C} \parallel p\rangle \rightarrow \bot \qquad \langle \{\underline{T_2}, \ldots, T_n\} \parallel \mathcal{C} \parallel \textbf{repeat}\, p\ \textbf{until}\ \ \phi\rangle \rightarrow \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c}{\langle \{\underline{T_1}, \ldots, T_n\} \parallel \mathcal{C} \parallel \textbf{repeat}\, p\ \textbf{until}\ \ \phi\rangle \rightarrow \langle \{T_1\} \cup A \parallel \mathcal{C} \parallel \epsilon\rangle : \bot}$$

$$\frac{\begin{array}{ll}\langle \{\underline{T_1}\} \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle \{T'_1, \ldots, T'_m\} \parallel \mathcal{C} \parallel \epsilon\rangle : c & \langle \{\underline{T'_i}, \ldots, T'_m, T_2, \ldots, T_n\} \parallel \mathcal{C} \parallel \\ \{T'_1\} \models \phi\ \ldots\ \{T'_{i-1}\} \models \phi & \textbf{repeat}\, p\ \textbf{until}\ \ \phi\rangle \rightarrow \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c'\end{array}}{\langle \{\underline{T_1}, \ldots, T_n\} \parallel \mathcal{C} \parallel \textbf{repeat}\, p\ \textbf{until}\ \ \phi\rangle \rightarrow \langle \{T'_1, \ldots, T'_{i-1}\} \cup A \parallel \mathcal{C} \parallel \epsilon\rangle : \bot}$$

Note that the use of the **try** combinator directly under the **repeat** constructor without a condition always leads to infinite loops.

Cases:
The **cases** construct checks whether the currently selected task has a specific form by (1) syntactic matching and (2) invoking a Lisp function in the environment provided by the matcher. If so, a specific action is triggered. Otherwise, the next case is checked. Note that the *default* matcher always matches. If we want the **cases** construct always to succeed, we can introduce a case *otherwise* → Id, where *Id* stands for the identity strategy.

$$\frac{}{\langle A \parallel \mathcal{C} \parallel \textbf{cases}\ ; \rangle \rightarrow \bot} \qquad \frac{A \not\models \phi_1 \qquad \langle A \parallel \mathcal{C} \parallel \textbf{cases}\ \ \phi_2 \rightarrow p_2 \ldots \phi_n \rightarrow p_n; \rangle \rightarrow res}{\langle A \parallel \mathcal{C} \parallel \textbf{cases}\ \ \phi_1 \rightarrow p_1 \ldots \phi_n \rightarrow p_n; \rangle \rightarrow res}$$

$$\frac{A \models \phi_1 \qquad \langle A \parallel \mathcal{C} \parallel p_1\rangle \rightarrow \bot \qquad \langle A \parallel \mathcal{C} \parallel \textbf{cases}\ \ \phi_2 \rightarrow p_2 \ldots \phi_n \rightarrow p_n; \rangle \rightarrow res}{\langle A \parallel \mathcal{C} \parallel \textbf{cases}\ \ \phi_1 \rightarrow p_1 \ldots \phi_n \rightarrow p_n; \rangle \rightarrow res}$$

$$\frac{A \models \phi_1 \qquad \langle A \parallel \mathcal{C} \parallel p_1\rangle \rightarrow \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c}{\langle A \parallel \mathcal{C} \parallel \textbf{cases}\ \ \phi_1 \rightarrow p_1 \ldots \phi_n \rightarrow p_n; \rangle \rightarrow \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c@(\textbf{cases}\ \ \phi_2 \rightarrow p_2 \ldots \phi_n \rightarrow p_n;)}$$

Thenselect:
The **thenselect** operator is used to split the result A' of a strategy execution p_1 on an agenda A, and to apply a specified second strategy p_2 to each of the tasks of A'. After the separate execution of p_2 on each of the tasks of A' the resulting agendas have to be combined to an overall result agenda. Note that it can be the case that the agendas cannot be combined if shared meta-variables have been instantiated differently. To simplify matters we assume that this check is performed when combining the agendas using $\cup$.

The **thenselect** operator is internally defined by a combination of the already introduced **then** operator and an internal operator **split** which does the mentioned splitting and recombination of the results:

$$p_1\ \textbf{thenselect}\ p_2 \equiv p_1\ \textbf{then}\ (\textbf{split}\ p_2) \tag{10.38}$$

The **split** operator is recursively defined over the agenda to which it is applied.

$$\frac{}{\langle \{\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \langle \{\} \parallel \mathcal{C} \parallel \epsilon\rangle : \bot} \qquad \frac{\langle \{T_1\} \parallel \mathcal{C} \parallel p\rangle \rightarrow \bot}{\langle \{T_1, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \bot}$$

$$\frac{\langle \{T_1\} \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c \qquad \langle \{T_2, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \bot}{\langle \{T_1, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \bot}$$

$$\frac{\langle \{T_1\} \parallel \mathcal{C} \parallel p\rangle \rightarrow \langle A \parallel \mathcal{C} \parallel \epsilon\rangle : c \qquad \langle \{T_2, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \langle A' \parallel \mathcal{C} \parallel \epsilon\rangle : c'}{\begin{array}{l}\langle \{T_1, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle \rightarrow \langle A \cup A' \parallel \mathcal{C} \parallel \epsilon\rangle \\ : (A \cup c')@(c \cup eval(\langle \{T_2, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle))\end{array}}$$

The first three rules are simple and handle the atomic case and the case of failure. The last rule represents the interesting case in which results from the base case and the recursive case have to be combined. The underlying idea is as follows: First, compute a solution A for the first task T_1, and a solution A' for the remaining tasks $T_2, \ldots, T_n$, and combine both agendas to obtain the first combined solution $A \cup A'$. The remaining solutions that need to be considered are A combined with all solutions for $T_2, \ldots, T_n$ encoded in c', as well as a new solution for T_1 (given by c), combined with all solutions that can now be computed for $T_2, \ldots, T_n$ (note that these might be different due to different substitutions), encoded by $eval(\langle \{T_2, \ldots, T_n\} \parallel \mathcal{C} \parallel \mathbf{split}\, p\rangle)$.

10.1.4 Discussion

Our main goal in defining a new tactic language for the specification of procedural strategies was the introduction of a small intermediate language which is independent of the underlying programming language of the prover and which allows for the specification of tactics within the proof document, while being extendable at the same time. The latter goal is achieved by allowing the invocation of library functions from the intermediate language. Of course, this poses the questions whether the first goal could be met at all, and whether such an interface represents a good design choice.

We believe that there are essentially two design choices: (1) a simple limited language that provides external mechanisms for its extension as our language, or (2) the design of a general language which is as complex as a full programming language. This insight comes from our previous investigations in the context of proof planning, where the goal to provide a fixed language to encode preconditions of methods failed, as analyzed in [DJP06]: “It was envisioned that the language for preconditions would reach a fix point, but it turned out that new domains need new types of preconditions.”. Our experiences indicate that this interface needs only to be used to a very limited extent, and that the main part of the tactics is formulated within the proof document, which proved to be very convenient from our perspective (see Chapter 14). Additional conditions are optionally defined on terms or on sequents within the underlying programming language and therefore require only a very limited knowledge about the internal data structures.

The strategy language itself is compact and due to its small number of constructs easy to learn. The main idea was to see the specification process of a proof strategy as a two-staged process, consisting of (1) the selection of proof operators and (2) augmentation of these proof operators with control knowledge. The selection process is inspired by the query languages SQL, OQL and XPath, which are standard for querying structured knowledge such as relational/object-oriented data bases or XML. Indeed, we see the development graph as an object-oriented hierarchical data base, object-oriented in the sense that the different knowledge kinds correspond to different classes and the theories serve as a hierarchical structuring mechanism. Our query mechanism makes the knowledge selection process explicit, and therefore, also allows for reasoning about it. It has been analyzed that this is not the case in current proof planners, but highly desirable (see [DJP06] p. 7).

One main goal of our approach was to give a precise semantics to the language constructs and to describe their behavior in terms of the proof state, and backtracking behavior. Further parameters can easily be incorporated to the language, for example, the search mode used within the **solve** construct. Indeed, depth first search and breadth first search have been implemented already within this language. The event store $\mathcal{C}$ provides a convenient means to exchange control information between different tactics and can be used not only for backtrack conditions, but more generally for any kind of control information, such as the restriction of the applicability of proof operators to specific positions.

10.2 A Declarative Language for Declarative Strategies

This section introduces the notion of a declarative strategy by analogy to procedural strategies as a means to automate the declarative style of proof. The overall idea is to replace the procedural justification of procedural strategies by a declarative proof script which is inserted into the proof document after the strategy has been executed. As a starting point, it is useful to consider the simplest form of a procedural strategy, which when being executed performs a macro transformation of a proof state and justifies it by a sequence of procedural commands at a lower level, e.g., a sequence of kernel commands. The kernel commands present a partial proof of the overall conjecture.

By analogy, we want to understand a declarative strategy to be such a macro operator, which – in contrast to a procedural strategy – is justified by a declarative proof script, i.e., a structured sequence of commands in a declarative proof language, which may just be partial. To meet the requirement to support the document-centric approach, we additionally require a declarative tactic to be specified on top of the declarative proof language.

Let us point out that due to the nature of a declarative proof language, declarative strategies provide two different forms to express partial proof scripts: Similar to procedural strategies, a declarative strategy transforms a goal to (simpler) goals. As these goals still need to be solved, the generated justification is only partial. While the transformation is fully specified within the procedural style, the declarative style allows the author of the tactic to be unspecific about the exact transformation. Therefore, additional proof obligations can arise due to gaps justifying intermediate statements of the script, for example, if the parameters **by** or **from** are not specified. Similar to proof methods in proof planning, these proof obligations require an expansion which can naturally be postponed.

Before introducing the details of the new language, let us further develop the basic underlying ideas and motivate it by several examples.

Towards A Declarative Induction Tactic

Consider the problem of showing the commutativity of addition in Presburger arithmetic, that is, $a+b=b+a$. Starting by induction over one variable, two possible proof attempts in the declarative style are shown in Listing 9. Note that both proofs are partial, as they contain unjustified statements. They can thus be seen as a proof sketch or a proof plan.

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals
    subgoal 0 + b = b + 0
    subgoal Suc a + b = b + Suc a
      using IH:a+b=b+a
  qed by (induct a)
qed
```

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals
    subgoal a + 0 = 0 + a
    subgoal a + Suc b = Suc b + a
      using IH: a+b=b+a
  qed by (induct b)
qed
```

Listing 9: Declarative proof with gap resulting by induction over a, respectively b

We want to automate the transformation of the initial conjecture to one of the proof states shown in Listing 9 by means of a declarative tactic. To that end, three steps are necessary: First, we need the *control information* over which variable the induction has to be performed. Note that there are two possibilities of how this information can be obtained: The first possibility is to pass the desired induction variable as a parameter to the strategy. The second possibility is to automatically compute all variables over which an induction can be performed and to select the most promising one, for example, by preferring those in recursion position. While the first possibility is usually preferred in an interactive setting, the second is needed for full automation. Depending on the selected induction variable, we either obtain the left or the right variant of the proof script. Analyzing both target scripts, it is immediately clear that a static proof script is not sufficient, as the concrete statements depend both on the considered problem, as well as on the parameter of the induction variable. However, the proof commands are the same in both situations. Therefore, the second step towards the declarative strategies consists in *abstracting* over the concrete statements of the target script and to replace them by schematic variables. This way, we obtain a more general schematic proof script, which uses the schematic variables as placeholders for the concrete statements that are inserted at runtime. The abstraction process results in a proof script which can be instantiated to both variants by instantiating the corresponding variables accordingly.

Finally, we have to provide a mechanism to instantiate the schematic variables of the schematic proof script at runtime by dynamically accessing the proof state. Similar to the original idea of proof planning methods, our solution consists of attaching a precondition to the strategy in the form of a declarative matcher and to use this precondition to compute the concrete shape of the statements of the proof script. At runtime, the declarative precondition is matched against the proof state, thereby making the relevant parts of the proof state explicit and providing variable bindings which can then be used in the body of the tactic.

Two possible realizations are shown in Listing 10. Both strategies consist of two parts: A meta-level precondition, marked with a shaded background, and a schematic proof script. Within the programming language, which is used in the **where** clause of the expression, we use double quotes to refer to expressions in the strategy language.

The simple example illustrates already the main features of our approach:

- Declarative strategies can be used both in an interactive as well as in an automated setting. Within the interactive setting, declarative strategies are manually executed as procedural strategies. However, in contrast to the procedural approach, the result

```
strategy natinduct
case * ⊢ P x
  where  (isinductvar "x")
->
proof
  subgoals by (induct x)
    subgoal P 0
    subgoal P (suc x) using IH: P
x
  end
qed
```

```
strategy natinduct
cases * ⊢ P x
  where (isinductvar "x")
->
proof
 L1:  P 0
 L2:  assume P x thus P (suc
x)
 P x from L1,L2 by(induct x)
qed
```

Listing 10: Declarative induction tactic in backward and forward style

of a strategy application is not only a new proof state, but also a declarative proof script, which is inserted at the point where the strategy has been invoked. This way, one obtains a readable result without the need of having to write the complete procedural proof script.

- The **by** parameter can be used to specify a particular proof strategy to be used for the expansion of a resulting proof obligation. Thus, it expresses the future in the form of a continuation.

- Stylistic choices, such as whether to prefer forward or backward style, as well as the granularity of the resulting proof script are explicitly controlled within the strategy. This is in contrast to translation based approaches, where such a control is only indirectly possible. Moreover, formulas can explicitly be named and the names be used to refer to these formulas.

- Declarative strategies can be expressed with the proof document as an extension of a declarative proof language. They are thus in the spirit of the document-centric approach.

Integration of a Computer Algebra System

One important feature of ΩMEGA's previous proof planner is the use of external computer algebra systems to have additional proof support for algebraic computation (see [KKS98] for details). This is because all algorithms that have been implemented in the CAS are directly available within the proof assistant without the need to reimplement them. Proof planning methods showed particularly useful to realize the integration of external computations, as they allowed the direct integration of the computation and to postpone its verification to a later time. We subsequently show how such an integration can be realized within our declarative strategy language: As noted before, schematic variables can be used as placeholder for arbitrary terms, in particular terms generated by an oracle without justification.

Listing 11 shows a declarative tactic that employs a CAS to transform a goal of the form `abs(GOALLHS)<GOALRHS` to `abs(Y)<GOALRHS`, where `Y` is the factorization of `GOALLHS`. To that end, `GOALLHS` is translated to the syntax of the CAS MAXIMA, and the result of the factorization translated back and bound to the schematic variable `Y`. If this succeeds, the script specified in the **proof** . . . **qed** block is instantiated and inserted. Moreover, the following proof plan is inserted in the document: (1) show the equality between `Y` (the

```
strategy maximafactorabs
cases
  * ⊢ ((abs(GOALLHS)) < GOALRHS) ->
  proof
    subgoal abs(Y) < GOALRHS by
    proof
      L2:(Y = GOALLHS) by abeliandecide
      L3:  abs(Y) = abs(GOALLHS) by (f=abs in arg_cong)
from L2
      trivial from L3
    qed
  qed
  with Y = (maxima-factor "GOALLHS")
```

Listing 11: Call of a CAS to factor a subterm of the goal formula

factorization provided by the CAS) and `GOALLHS`, (2) use the fact that functions yield the same value on the same arguments. Note that the same tactic is also expressible in a forward style by relying on **assume** and that all labels in the proof script are generated at runtime and are renamed if already present in the context.

Factorization is a typical example for a computation for which checking of a solution is much simpler than finding it: Given a term and its factorization, we only need to multiply out the factors of the factorization and to compare the result with the original term, possibly rearranging the subterms.

Complex Estimate

Finally, let us show how more complicated heuristics can be expressed in the declarative style. To that end, we consider one of the so-called *limit heuristics* which were originally proposed as a challenge by W. Bledsoe:

"When trying to use a hypothesis of the type

$$|A| < E'$$

(and possibly other hypothesis) to establish a conclusion of the type

$$|B| < E$$

first try to find a substitution σ which will allow B_σ to be expressed as a non-trivial combination of A, $(B = K \cdot A + L)$, and then try to establish the three new conclusions

1. $(|K| < M)_\sigma$ *for some M*
2. $(|A| < (E/(2 \cdot M))_\sigma$
3. $(|L| < E/2)$

Such a procedure is valid because if we can indeed find such a σ and prove A,

B, and C, then we would have

$$\begin{aligned}|B|_\sigma &= |K \cdot A + L|_\sigma \\ &\leq (|K| \cdot |A| + |L|)_\sigma \\ &< M \cdot E/(2 \cdot M) + E/2 \\ &= E\end{aligned}$$

Of course, this is based on the triangle inequality, and uses the fact that $1/2 + 1/2 = 1$, $M \cdot 1/M = 1$ for $M > 0$." *(see [BBH72] page 13)*

This strategy was also used in the proof planner Multi and realized in the method ComplexEstimate (see [Mel98b] for the first implementation of this idea), which relied on a computer algebra system to find the linear combination. In the previous Ωmega system, the method was realized by 113 lines of Lisp code, even though it could be expressed more concisely using frames, as shown in Listing 12.

Method: ComplexEstimate				
premises	L1, ⊕L2,⊕L3,⊕L4			
conclusions	⊖L17			
appl. cond.	$\exists\sigma.\texttt{GetSubst}(a, b) = \sigma$ **and** $\exists k, l.\texttt{CASsplit}(a\sigma, b) = (k, l)$			
proof schema	L1.	Δ	$\vdash \lvert a\rvert < \epsilon_1$	(assumption)
	L2.	Δ	$\vdash \lvert k\rvert \leq \boldsymbol{M}$	(open)
	L3.	Δ	$\vdash \lvert a\sigma\rvert < \epsilon/(2 * \boldsymbol{M})$	(open)
	L4.	Δ	$\vdash \lvert l\rvert < \epsilon/2$	(open)
	L5.	Δ	$\vdash b = b$	(ax)
	L6.	Δ	$\vdash b = k * a\sigma + l$	(CAS;L5)
	L7.	Δ	$\vdash 0 < \boldsymbol{M}$	(open)
	L8.	Δ	$\vdash \lvert b\rvert \leq \lvert k * a\sigma\rvert + \lvert l\rvert$	(triang;L6)
	L9.	Δ	$\vdash \lvert b\rvert \leq \lvert k\rvert * \lvert a\sigma\rvert + \lvert l\rvert$	(Mval;L8)
	L10.	Δ	$\vdash \lvert k\rvert * \lvert a\sigma\rvert + \lvert l\rvert \leq M * \lvert a\sigma\rvert + \lvert l\rvert$	(mult≤;L2)
	L11.	Δ	$\vdash \lvert b\rvert \leq M * \lvert a\sigma\rvert + \lvert l\rvert$	(trans≤;L9,L10)
	L12.	Δ	$\vdash M * \lvert a\sigma\rvert < M * \epsilon/(2 * M)$	(mult<;L3)
	L13.	Δ	$\vdash M * \lvert a\sigma\rvert + \lvert l\rvert < M * \epsilon/(2 * M) + \lvert l\rvert$	(add<;L12)
	L14.	Δ	$\lvert b\rvert < M * \epsilon/(2 * M) + \lvert l\rvert$	(trans<;L11,L13)
	L15.	Δ	$\vdash M{*}\epsilon/(2{*}M){+}\lvert l\rvert < M{*}\epsilon/(2{*}M){+}\epsilon/2$	(add<;L4)
	L16.	Δ	$\vdash \lvert b\rvert < M * \epsilon/(2 * M) + \epsilon/2$	(trans<;L14,L15)
	L17.	Δ	$\vdash \lvert b\rvert < \epsilon$	(fix; L6,L7,L1 L2,L3,L4)

Listing 12: Abstract representation of the method ComplexEstimate

Let us remark that even though the frame representation of Listing 12 is declarative, it cannot easily be understood by non-experts. This is because the representation is totally flat and the dependencies between the different proof lines can only be resolved by following the labels and their links. This also explains why minor imperfections have not been recognized: The proof lines `L5` and `L7` are not needed, as they are not used within the proof schema. Moreover, `L7` is implied by `L2`, which can only be closed if $M > 0$. Let us point out that the implementation relies on many other (manually encoded) proof methods, namely exactly those mentioned in the rightmost column of Listing 12.

Subsequently, we show the reimplementation of the method within the new proof language in Listing 13, in which we furthermore add the line $|b| \leq |k * a + l|$. Within the

```
strategy ComplexEstimate
cases
  |a| < e_1,* ⊢ |b| < ε
  where (not (terms-are.equal "k" "0"))
  with σ in (findaddsubst "a" "b")
    (k,l) = (maxima-divide "b" (substapply σ
"a")) ->
  proof
    put σ
    L2: |k| ≤ M
    L3: |a| < ε/(2 * M)
    L4: |l| < ε/2
    L17: |b| < ε
    proof
      b = k * a + l by CAS
      |b| ≤ |k * a + l| by abs
      .≤ |k * a| + |l| by triang
      .≤ |k| * |a| + |l| by Mval
      .≤ M * |a| + |l| from L2 by trans≤
      .< M * ε/(2 * M) + |l| from L3 by trans≤
      .=ε/2 + |l| by arith
      .<ε/2 + ε/2 from L4 by trans<
      .=ε
    qed
  qed
```

Listing 13: ComplexEstimate in the new strategy language

new strategy language, the heuristic can be expressed in a concise way: It consists only of 24 lines of code in a declarative proof language, compared to 113 lines of Lisp code of the old method. Moreover, due to the automatic management of the contextual information and the hierarchical structure imposed by the proof language, the method is much more readable. In particular, the inequality chain can easily be understood due to the support of inequality reasoning in the proof language.

10.2.1 Syntax of the Basic Language

Let us now turn to the concrete syntax of the declarative strategy language. As mentioned above, declarative proof strategies are defined in an extension of the declarative proof language. The extended language provides additional constructs to specify the applicability of the strategy and provides mechanisms to bind schematic variables, as well as to evaluate meta-level conditions. The grammar of the basic language is shown in Figure 10.3.

Similar to the procedural strategy language, it comes along with a **case** construct and matching facilities to relate schematic variables with a given proof state and to restrict the applicability of the strategy. Syntactic conditions can be extended by meta-level restrictions in the form of a **where** construct. As the matching might not be unique, a matcher explicitly introduces choice points during the proof search process. The declarative proof language is linked in by the non-terminal <proof> (see Section 8.3). Moreover, the formula parser is linked in using the non-terminal < form >. Let us stress here that we support underspecified proof scripts, which are obtained by leaving out **by**, **from** or

<defstrat>	::= **strategy** <name> <stratexp>
<stratexp>	::= **cases** (<matcher> <stratexp>)$^+$;
	$\mid$ <proof> (**with** <assignments>)?
<matcher>	::= <matchhead> <whereexp>?
	(**with** <assignments>)?
<whereexp>	::= **where** <prog>
<matchhead>	::= <sequent> $\mid$ var
<sequent>	::= <termpatterns> (**,***)? $\vdash$ <termpattern>
<termpatterns>	::= <termpattern> (**,** <termpattern>)*
<termpattern>	::= <form> $\mid$ [<term>] (<termqualifier>)?
<termqualifier>	::= + $\mid$ − $\mid$ <var>
<assignments>	::= <lhs> <assignop> <prog>
<assignop>	::= **=** $\mid$ **in**
<lhs>	::= <form> $\mid$ **(** <form> (<,> <form>)$^+$ **)**

Figure 10.3: Basic Tactic Language

both.

Schematic variables that are not matched in the precondition can explicitly be computed within the language construct **with**. Currently, the language provides two assignment operators, a deterministic assignment =, and a non-deterministic operator **in** that chooses from a list of possible values. As we cannot expect to provide a fixed language to express meta-level conditions and to perform meta-level analysis (e.g., the extraction of the admissible induction variables as in Listing 10), a reasonable strategy is to link-in the underlying programming language of the prover here, indicated in the grammar by <prog>.

Parameter Passing

For strategies it is often convenient to pass control information in the form of arguments when calling the tactic. In the introductory example (see Listing 9), we invoked the tactic `induct` with the argument "x" indicating the induction position. A similar mechanism is desirable in the case of declarative tactics. In our language, arguments are treated as schematic variables. If a schematic variable occurs in the proof script, but is neither used in the **case** construct nor bound within the **with** environment, it corresponds to a required argument. Schematic variables that are computed within the tactic can be passed as optional arguments. In such a case, the passed argument overwrites the computed argument. We provide the common syntax *var=value* **in** *tactic*.

10.2.2 Semantics

Figure 10.4 shows the semantics of our language constructs by showing how to expand a declarative tactic to a declarative proof script. To keep the presentation simple, we omit the explicit management of backtracking points in the form of a continuation and present the semantics of the language by a non-deterministic expansion mechanism. The expansion mechanism works on configurations $\langle PS; \Gamma; exp\rangle$, where PS denotes the current proof state, and Γ a context, which is initially empty and keeps track of bindings for schematic variables. *exp* denotes the expression to be expanded. Configurations evaluate either to a proof script, denoted by the relation $\hookrightarrow$, or to an environment, denoted by the relation $\rightarrow$. We use the notation $\Gamma \cup a = b$ to denote the update of Γ with the binding $a = b$, and the symbol $\perp$ to denote failure. To keep the rules simple, some rules are non-deterministic. In the actual implementation, of course, all results are lazily produced and stored for backtracking, as explained in the previous section. $\mathsf{instance}(\Gamma, S)$ denotes the instantiation of the schematic proof S by replacing the schematic variables with their values in Γ. It is only applicable if all schematic variables of S are bound. We use eval to evaluate a Lisp expression < prog >; the sequent matching is abstracted in the function match (which is also non-deterministic).

To enhance readability, we have grouped corresponding rules together. The first group describes the expansion of the **case** construct, which returns the result of the first case that succeeds. An individual case is either a proof script (second group), or of the form < matchhead (**where** exp)? > (third group). The value of schematic variables is computed within the **with** construct (see the last group) which uses eval to evaluate expressions of the underlying programming language. Sequent matching works by first invoking the matcher on the current proof state and then evaluating the additional condition.

10.2.3 Extension of the Basic Language by Dynamic Patterns

The declarative tactic language presented above is already sufficient to encode a variety of common patterns of reasoning. However, compared to its procedural counterpart, it is more restricted, as dynamic structures cannot be modeled. As a matter of fact, there exist powerful procedural strategies using for example the constructs of loops, such as simplification, which are per se difficult to express declaratively, as we cannot expect to determine their result unless we execute it. This is not problematic in general, as we can always invoke a procedural strategy within the **by** construct of the declarative language. However, there are also situations where it is highly desirable to capture dynamic structures at the declarative level and to express a dynamic proof pattern subsequently. *Dynamic patterns* have been designed to cover dynamic structures, such as finite sums, factorizations, and to make the involved sums/factors available in the context. They can be used both in a precondition of a **case** construct, but also to destructure the result of an external computation within a **with**. This way, it is possible to recover the structure of a result given by an external computation, such as a factorization, and to express how to process their results further in the form of a continuation. For example, all we know about the result term of factorization in Figure 11 is that it is of the form `Y`. Using a dynamic pattern, we obtain back the number of the factors and can trigger an action for each of the factors. To be able to iterate over the resulting term list, an iteration concept **foreach** is introduced. Note that while schematic variables keep the number of proof lines fixed and abstract over the concrete statements, dynamic patterns in combination with iteration allow for dynamic proof scripts.

We give two examples. Consider the following problem which is taken from [JKK+05]. Given the goal

$$3 + f(x) + g(x, y) = x + g(x, y) + h(y, x)$$

we want to write a tactic that cancels common summands to obtain

$$3 + f(x) = x + h(y, x)$$

The example is a typical instance for a situation in which the goal state has a rather simple dynamic structure, i.e., a finite sum of terms, and a dynamic destructuring of it is sufficient to compute a desired successor goal state, i.e., a proof state where all common

$$\frac{\langle PS;\Gamma;c_1\rangle \hookrightarrow \bot \qquad \langle PS;\Gamma;\mathbf{case}\ c_2\dots c_n\rangle \hookrightarrow S}{\langle PS;\Gamma;\mathbf{case}\ c_1\dots c_n\rangle \hookrightarrow S} \qquad \frac{}{\langle PS;\Gamma;\mathbf{case}\ \epsilon\rangle \hookrightarrow \bot}$$

$$\frac{\langle PS;\Gamma;c_1\rangle \hookrightarrow S}{\langle PS;\Gamma;\mathbf{case}\ c_1\dots c_n\rangle \hookrightarrow S}\ S \neq \bot$$

- -

$$\frac{\langle PS;\Gamma;ass\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{proof}\ \mathbf{with}\ ass\rangle \hookrightarrow \bot} \qquad \frac{\langle PS;\Gamma;ass\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma';\mathit{proof}\rangle \hookrightarrow S}{\langle PS;\Gamma;\mathit{proof}\ \mathbf{with}\ ass\rangle \hookrightarrow S}\ S \neq \bot$$

$$\frac{}{\langle PS;\Gamma;\mathit{proof}\rangle \hookrightarrow \mathsf{instance}(\Gamma,\mathit{proof})}$$

- -

$$\frac{\langle PS;\Gamma;\mathit{matcher}\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{matcher\ stratexp}\rangle \rightarrow \bot} \qquad \frac{\langle PS;\Gamma;\mathit{matcher}\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma';\mathit{stratexp}\rangle \hookrightarrow S}{\langle PS;\Gamma;\mathit{matcher\ stratexp}\rangle \hookrightarrow S}$$

$$\frac{}{\langle PS;\Gamma;\mathit{matchhead}\rangle \rightarrow \mathsf{match}(\mathit{matchhead},PS)} \qquad \frac{\langle PS;\Gamma;\mathit{matchhead}\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{matchhead}\ \mathbf{where}\ \mathit{exp}\rangle \rightarrow \bot}$$

$$\frac{\langle PS;\Gamma;\mathit{matchhead}\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma';\mathbf{where}\ \mathit{exp}\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{matchhead}\ \mathbf{where}\ \mathit{exp}\rangle \rightarrow \bot}$$

$$\frac{\langle PS;\Gamma;\mathit{matchhead}\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma;\mathbf{where}\ \mathit{exp}\rangle \rightarrow \Gamma''}{\langle PS;\Gamma;\mathit{matchhead}\ \mathbf{where}\ \mathit{exp}\rangle \rightarrow \Gamma''}$$

$$\frac{\langle PS;\Gamma;ass\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma';\mathsf{eval}(c)\rangle \rightarrow \top}{\langle PS;\Gamma;\mathbf{where}\ c\ \mathbf{with}\ ass\rangle \rightarrow \Gamma'} \qquad \frac{\langle PS;\Gamma;ass\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathbf{where}\ c\ \mathbf{with}\ ass\rangle \rightarrow \bot}$$

$$\frac{\langle PS;\Gamma;ass\rangle \rightarrow \Gamma' \qquad \langle PS;\Gamma';\mathsf{eval}(c)\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathbf{where}\ c\ \mathbf{with}\ ass\rangle \rightarrow \bot} \qquad \frac{\langle PS;\Gamma;c\rangle \rightarrow \top}{\langle PS;\Gamma;\mathbf{where}\ c\rangle \rightarrow \Gamma}$$

- -

$$\frac{\langle PS;\Gamma;\mathsf{eval}(\mathit{prog})\rangle \rightarrow c \qquad \langle PS;\Gamma \cup \mathit{lhs} = c;\mathit{ass}'\rangle \rightarrow \Gamma'}{\langle PS;\Gamma;\mathit{lhs}\ = \mathit{prog}\ \mathit{ass}'\rangle \rightarrow \Gamma'}\ c \neq \bot$$

$$\frac{\langle PS;\Gamma;\mathsf{eval}(\mathit{prog})\rangle \rightarrow [c_1,\dots,c_n] \qquad \langle PS;\Gamma \cup \mathit{lhs} = c_i;ass\rangle' \rightarrow \Gamma'}{\langle PS;\Gamma;\mathit{lhs}\ \mathbf{in}\ \mathit{prog}\ \mathit{ass}'\rangle \rightarrow \Gamma'}\ n \geq 1 \land 1 \leq i \leq n$$

$$\frac{\langle PS;\Gamma;\mathsf{eval}(\mathit{prog})\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{lhs\ assignop\ prog\ ass}'\rangle \rightarrow \bot}$$

$$\frac{\langle PS;\Gamma;\mathit{eval}(\mathit{prog})\rangle \rightarrow c \quad \langle PS;\Gamma \cup \mathit{lhs} = c;\mathit{ass}'\rangle \rightarrow \bot}{\langle PS;\Gamma;\mathit{lhs} = \mathit{prog}\ \mathit{ass}'\rangle \rightarrow \bot}\ c \neq \bot$$

Figure 10.4: Expansion Rules for a Declarative Tactic

summands have been canceled. Interestingly, even though the structure is dynamic, syntactic patterns are commonly used in mathematical practice to capture such a structure, by making use of ellipses (dot notation). In our example, a dynamic number of summands can be expressed by the pattern `A_1 + ...+ A_N`. Internally, patterns are implemented by subsequently invoking the matcher for pattern A until it fails, taking the associativity of the binary operator into account, resulting in a list of matches which are stored in an internal variable A.

```
strategy cancelsum
cases * |- A_1 + ..  + A_N = B_1 + ..  + B_M
 proof
  L1:  C_1 + ..  + C_N = D_1 + ..  + D_M
  A_1 + ..  + A_N = B_1 + ..  + B_M from L1  qed
 with
  foreach I in 1..N where (not (member ``A_I'' ``B'')) C_I
= A_I
  foreach I in 1..M where (not (member ``B_I'' ``A'')) D_I
= B_I
```

Listing 14: Illustrating dynamic patterns

Listing 14 shows a strategy that uses dynamic pattern matching constructs `A_1 + .. + A_N` and `B_1 + ..  + B_M` to capture the finite sum. Internally, such structures are destructured, and its members stored in a corresponding list `A` and `B`. Based on these lists, new dynamic constructs `C` and `D` are constructed, which correspond to filtered versions of `A` and `B`, respectively. The filtered lists have the same length than the original lists, but contain error elements at those positions at which the filter function failed. During term construction, the failure elements are removed.

As second example, we consider an extension of the declarative strategy `maximafactorabs` in the context of the so-called *limit domain*. One heuristic of the limit domain to bound factors is to reduce the problem that the product $\beta\gamma$ is arbitrarily small to the problem of showing that β is arbitrarily small and γ can be bounded. This heuristic is called *factor bounding* and described (in [Bee98], p. 77f) as follows :

"The following rule is stated for simplicity using only two factors, but the rule is implemented for a product of any number of factors.

$$\frac{\Gamma, |\alpha| < \delta \vdash \gamma < M \qquad \Gamma, |\alpha| < \delta| \vdash |\beta| < \epsilon/(M+1)}{\Gamma, |\alpha| < \delta \vdash |\beta\gamma| < \epsilon}$$

When this rule is implemented, we take M to be a fresh meta-variable, and forbid to M all the variables that are forbidden to δ. In the present implementation, the rule is used only when δ is a meta-variable."

While factorization has already been considered in Listing 11, dynamic patterns allow us to further analyze the result of the factorization and to express the factor bounding as the continuation of a successful factorization.

The grammar for the extended language constructs is shown in Figure 10.5. Binary patterns (< binoppat >) can be used in places where previously only < form > was allowed. Proof steps (< step >) are extended by the < foreachstep > construct, and assignments by the foreach assignment (< foreachass >). These extensions will allow us to specify a variant of the factorbound method in a convenient way (see Listing 15).

```
<binoppat>      ::= <pattern> <binop> .. <binop> <pattern>
<listaccess>    ::= (<listterm> | <var>) _ (<var> | <number>)
<listterm>      ::= <listdel> .. <listdel>
<listdel>       ::= <var> | <number> | <pattern>
<pattern>       ::= <binoppat> | <listaccess> | <from>
<foreachexp>    ::= foreach <var> in <listterm>
                    (where cond)?
<foreachstep>   ::= <foreachexp> <steps> end
<foreachass>    ::= <foreachexp> <var> _ <var>=<prog>
```

Figure 10.5: Dynamic matching constructs

Expression	Meaning
`A_1 + .. + A_N`	finite sum with N summands
`abs(A_1) * .. * abs(A_N)`	product with N factors of the form `abs(_)`
`(X_1 + Y_1) * .. * (X_N + Y_N)`	product of terms of binary sums
`1 .. 5`	list [1,2,3,4,5]
`abs(A_1) .. abs(A_N)`	list with N terms of the form `abs(_)`

Figure 10.6: Patterns using ellipses

Ellipses.

So far our constructs for matching and constructing terms are static in the sense that their actual form was already determined at compile time. For example, a pattern of the form $lhs = rhs$ checks whether the input formula is an equality and binds its first argument to lhs and its right argument to rhs. *Dynamic Patterns* on the contrary are patterns that capture dynamic structures, such as all elements of a finite list. We support a simple dynamic pattern, an ellipsis for binary operators, written $A\ op \ldots op\ A'$, which acts like a Kleene star, as well as a list pattern which is similar except that < op > is omitted. Internally, such dynamic patterns are represented as lists, whose length is stored in an additional variable. To individually access the lists, we provide an accessor function _. That is, A_n denotes the n-th element in the list A. If n is a variable, then n is called *access variable.* In the current implementation, patterns are restricted to *simple patterns*, which are patterns that unify under a substitution σ whose domain consists only of access variables. Patterns can be used both in conditions, as left-hand side of assignments, as well as in proof script terms. Some examples are shown in Figure 10.6.

The foreach construct

The **foreach** construct provides a simple form of iteration over a list of values obtained from a dynamic pattern. It can be used to construct statements in the proof script language as well as to construct a list of schematic variables. Its expansion rules are shown in Figure 10.7, grouped into the expansion rules to expand **foreach** within a proof script, and the expansion rules to expand **foreach** in assignments. Note that in case of assignments a list containing all produced values is constructed, which has always the length of the list over which it is iterated. In the case that the condition evaluates to $\perp$ a term *false* is inserted at the corresponding position.

```
strategy factorbound
cases
   abs(LHS)<RHS,* ⊢ abs(GOALLHS) < GOALRHS
   where (and (variable-eigenvar.is "GOALRHS")
                (metavar-is "RHS")
                (some #'(lambda (x) (term= "LHS" "x")) "Y_1 ..  Y_N"))
   with Y_1 * ..  * Y_N = (maxima-factor "GOALLHS")
         j = (termposition "LHS" "Y_1 ..  Y_N")
   ->
   proof
     L1:  GOALLHS= Y_1 * ..  * Y_N  by abeliandecide
     foreach i in 1..N where (not (= "j" "i"))
       Y_j <= MV_j by linearbound
     end
     L2:  abs(GOALLHS)=abs( Y_1 * ..  * Y_N ) from L1
     .<= abs(Y_1) * ..  * abs(Y_N)
     .< MV_1 * ..  * MV_N
     .<= GOALRHS
   qed
   with foreach i in 1..N
         M_i = (if (= "i" "j") "RHS" (make-metavar (term-type "RHS")))
```

Listing 15: Dynamic pattern matching and proof script generation

Illustration of the Strategy

We now illustrate our new strategy by means of an example. We consider the problem of proving $\lim_{x\to 3}\frac{x^2-5}{x-2}=4$. After expanding the definition of lim, the proof state consists of the two goals $\epsilon>0, |x-3|<?\delta \vdash |\frac{x^2-5}{x-2}-4|<\epsilon$ and $\epsilon>0\vdash ?\delta>0$. The declarative proof script is shown at the top of Listing 16, where the declarative tactic `factorbound` (see Figure 15) is not yet processed.

Processing the `factorbound`-statement expands it and results in the following steps:

1. The pattern of the cases condition is matched, yielding the following binding: $\{\texttt{LHS}\mapsto x-3, \texttt{RHS}\mapsto ?\delta, \texttt{GOALLHS}\mapsto \frac{x^2-5}{x-2}-4, \texttt{GOALRHS}\mapsto\epsilon\}$

2. To be able to evaluate the **where** condition, the first **with** part is evaluated. This results in the following factorization: $Y_1*\ldots*Y_n=(x-3)*(\frac{1}{x-2})*(x-1)$. Internally, a list $Y=[(x-3),(\frac{1}{x-2}),(x-1)]$ is generated, n is bound to 3. In the next assignment, j is bound to 1 by looking up $x-3$ in the list of factors.

3. The conditions of the **where** part evaluates to true

4. The **with** part of the **proof** is evaluated, generating a list $M=[?\delta, ?MV1, ?MV2]$ of length 3.

5. The **proof** part is expanded and inserted, resulting in the proof script shown at the bottom in Figure 16.

$$\frac{\langle PS;\Gamma;\mathit{listterm}\rangle \rightarrow [e_1,\ldots,e_n] \quad \begin{array}{l}\langle PS;\Gamma;\\ \quad \textbf{iterate}\ \texttt{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ (\textbf{where}\ c)?\mathit{exp} \hookrightarrow S\end{array}}{\langle PS;\Gamma;\textbf{foreach}\ \mathit{var}\ \textbf{in}\ \mathit{listterm}\ (\textbf{where}\ c)?\ \mathit{exp}\ \textbf{end}\ \rangle \hookrightarrow S}$$

$$\frac{}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [](\textbf{where}\ c)?\ \mathit{exp}\ \textbf{end}\ \rangle \hookrightarrow \epsilon}$$

$$\frac{\langle PS;\Gamma\cup \mathit{var}=e_1;\mathit{exp}\rangle \hookrightarrow S1\ \langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_2,\ldots,e_n]\ \mathit{exp}\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ \mathit{exp}\rangle \hookrightarrow S1\ S2}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup \mathit{var}=e_1;\mathit{exp}\rangle \hookrightarrow S1\\ \langle PS;\Gamma\cup \mathit{var}=e_1;c\rangle \rightarrow \top\end{array} \quad \langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_2,\ldots,e_n]\ \textbf{where}\ c\ \mathit{exp}\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ \textbf{where}\ c\ \mathit{exp}\rangle \hookrightarrow S1\ S2}$$

$$\frac{\langle PS;\Gamma\cup \mathit{var}=e_1;c\rangle \rightarrow \bot\ \langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_2,\ldots,e_n]\ \textbf{where}\ c\ \mathit{exp}\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ \textbf{where}\ c\ \mathit{exp}\rangle \hookrightarrow S2}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma;\mathit{listterm}\rangle \rightarrow [e_1,\ldots,e_n]\\ \langle PS;\Gamma;\textbf{iterate}\ \texttt{ass}\ \textbf{in}\ [e_1,\ldots,e_n]\ (\textbf{where}\ c)?\mathit{prog}\rangle \rightarrow \Gamma'\end{array}}{\langle PS;\Gamma;\textbf{foreach}\ \mathit{var}\ \textbf{in}\ \mathit{listterm}\ (\textbf{where}\ c)?\ \underbrace{\texttt{name\_}\mathit{var} = \texttt{prog}}_{=:ass}\rangle \rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup \mathit{var}=e_1;\mathit{ass}\rangle \rightarrow \Gamma'\\ \langle PS;\Gamma\cup \mathit{var}=e_1;c\rangle \rightarrow \top\\ \langle PS;\Gamma'\backslash(\mathit{var}=e_1);\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_2,\ldots,e_n]\ \textbf{where}\ c\ \mathit{ass}\rangle \rightarrow \Gamma''\end{array}}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ \textbf{where}\ c\ \mathit{ass}\rangle \rightarrow \Gamma''}$$

$$\frac{\langle PS;\Gamma\cup \mathit{var}=e_1;c\rangle \rightarrow \bot\ \langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_2,\ldots,e_n]\ \textbf{where}\ c\ \mathit{ass}\rangle \rightarrow \Gamma'}{\langle PS;\Gamma;\textbf{iterate}\ \mathit{var}\ \textbf{in}\ [e_1,\ldots,e_n]\ \textbf{where}\ c\ \mathit{ass}\rangle \rightarrow \Gamma'}$$

Figure 10.7: Expansion of the **foreach** construct

10.2.4 Discussion

While it has been recognized that the declarative style of a proof has many advantages over the procedural style, current state of the art proof assistants do not provide tools that generate declarative proof scripts or parts of them automatically. Instead, the complete proof script must be written manually by the user. The general techniques developed in this section provide new foundational ideas to overcome this deficiency. This contributes both to the community of interactive theorem proving and the community of proof planning, as they can both be used in an interactive, as well as in an automatic setting. We have shown how to automate the declarative style of proof by introducing the notion of a declarative strategy, which is defined as an extension of a declarative proof language and combines the advantages of the procedural and declarative style of proof. As a consequence, proof strategies can be specified within the proof document in a language that the user is already familiar with. Stylistic choices, such as the granularity of the resulting

```
theorem th1:  $\lim_{x\to 3}\frac{x^2-5}{x-2} = 4$
proof
  subgoals
    subgoal $|\frac{x^2-5}{x-2} - 4| < \epsilon$ using A1:$\epsilon > 0$ and A2:$|x-3| <?\delta$ by factorbound
    subgoal $?\delta > 0$ using $\epsilon > 0$
  end by limdefbw
qed
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
theorem th1:  $\lim_{x\to 3}\frac{x^2-5}{x-2} = 4$
proof
  subgoals
    subgoal $|\frac{x^2-5}{x-2} - 4| < \epsilon$ using A1:$\epsilon > 0$ and A2:$|x-3| <?\delta$
    proof
      L1:$\frac{x^2-5}{x-2} - 4 = (x-3) * (\frac{1}{x-2}) * (x-1)$ by abeliandecide
      $|x-1| \leq ?MV1$ by linearbound
      $|\frac{1}{x-2}| \leq ?MV2$ by linearbound
      L2:  $|\frac{x^2-5}{x-2} - 4| \leq |(x-3) * (\frac{1}{x-2}) * (x-1)|$ from L1
      $. \leq |x-3| * |\frac{1}{x-2}| * |x-1|$
      $. < ?\delta * ?MV1 * ?MV2$
      $. \leq \epsilon$
    qed
    subgoal $?\delta > 0$ using $\epsilon > 0$
  end by limdefbw
qed
```

Listing 16: Declarative proof script of the example before and after processing the call of the declarative tactic `factorbound`

proof scripts, naming of important formulas that are expected to play a major role in the subsequent proof, or the choice between forward and backward steps can easily be expressed within the language. Therefore, declarative strategies contain additional information that cannot easily be expressed within the procedural strategies. Moreover, they allow for a new kind of subgoal resulting from the refinement of proof steps. As a consequence, they are particularly suited to express abstract proof plans, which, for the first time in the literature, is expressed as a partially specified declarative proof script with a precise semantics. Of course, this does not mean that all strategies must be declarative: It is the combination of declarative and procedural strategies which makes the approach powerful.

From a proof planning perspective, we have established a new format to encode common patterns of reasoning in the form of declarative strategies, which is a main issue when trying to automate human problem solving behavior. In particular, declarative strategies provide a major means for global search control in the spirit of proof planning. The fact that the majority of the knowledge is declaratively encoded in an intermediate language within the proof document is a major step to increase the usability of proof planners by users, as well as to provide a possibility to evaluate proof planning systems, as all knowledge becomes explicit. We have shown how it is possible to capture dynamic structures in the declarative style in the form of dynamic patterns, and how to express a dynamic continuation subsequently. At the same time, we support local control knowledge to be incorporated. From a conceptual point of view, the separation of the strategy language from the programming language plays an essential role in the overall design of our system. It provides benefits both for the maintenance of proof planning systems, but also for the runtime behavior, as optimizations can easily be incorporated and local decisions are treated locally. A key design of the language is to provide general constructs and combinators to encode common patterns of reasoning, while providing a possibility to encode local heuristics locally in the underlying programming language.

10.3 Related Work

To the best of our knowledge, no equivalent tactic language exists that combines all our features of our approach. In particular, there is no tactic language supports queries that take advantage of the theory structure. To allow for a detailed comparison with existing approaches, we therefore consider the following three components of our language separately: The query language used within procedural strategies, the declarative language to encode procedural strategies, and finally, the declarative language to encode declarative strategies.

10.3.1 Math Search and Mathematical Knowledge Retrieval

The need for querying mathematical repositories came up when they became large and their maintenance important. Searching and retrieving of mathematical knowledge in these data bases has been studied for several proof assistants. We subsequently present a selection of related approaches that appear to be the most important ones.

MML Query

MML Query [BR03] addresses the problem of querying knowledge from the Mizar Mathematical Library (MML) and to present this knowledge in a readable format to the user in the form of a hyperlinked HTML document. Similar to our approach, the knowledge is classified into different categories, such as articles, definitions, theorems, function symbols, but also notations, which is currently not supported within our approach. The language is essentially based on extracting information from terms, such as their function symbols, and comparing the result with a desired outcome. While this is also possible within our language, pattern matching and the use of the resulting binding within the query is not supported in MML. Rather, predefined accessors are provided, such as the right-hand side of a definition, which can easily be defined in the form of a declarative pattern within our language. Distances between articles or theories are not considered within MML Query.

MoMM

MoMM [Urb06] is a tool for the fast retrieval of matching theorems from MML. It is used to (i) find theorems in the data base that complete a partial justification, i.e., complete a proof command φ **by** $A1, A2$ to φ **by** $A1, A2, Th$, and (ii) to check whether a proved theorem exists already in a more general form in the library. To that end, theorems and partial justifications are translated to clauses and a subsumption check is invoked. If it succeeds, a message is issued to the user, who in turn can complete the justification or remove the less general theorem from the library. Within our work, the functionality (i) can be achieved by querying all inferences that close the given proof situation, the process including the completion of the statement could even be automated. In contrast, queries in Mizar cannot be used within the proof checker, and the proof checker cannot be extended by user-defined strategies. Let us point out here that our proof scripts are more structured than those in MIZAR, as we have arguments **from** to specify a context, and **by** to specify a strategy/inference, while in MIZAR only a single construct **by** to collect facts and assumptions is used. The selection of subsuming theorems within the library ((ii)) has not yet been considered within our context, as our language is intended to support proof development rather than maintenance of the library. Due to the size of our libraries, we do not use sophisticated indexing techniques, which however, could easily be integrated within our framework as an optimization.

LSI-based Search in Alcor

In the context of the Alcor [Cai05] user interface for MIZAR, the technique of latent semantic indexing (LSI) has been applied (see [CG07]). Even though it is acknowledged by the authors that the success of the approach is difficult to judge and further refinements are needed, we list the approach here, because it represents an interesting approach that might become more relevant in the future. The main advantage of LSI over other approaches is that similar documents can be found, even if the concrete terms of the query do not occur in the document. Given a set of documents and a set of terms, LSI works by computing a matrix Δ, in which the entry a_{ij} corresponds to the relevance of term j in document i. Using the singular value decomposition, the similarities between a query term and a document can now be computed in a vector space of smaller (user defined) dimension. For an overview of LSI see [Ros00].

HELM, Whelp and MATITA

Whelp [AGC$^+$04] was developed as a semantic search engine for the COQ library in the context of HELM [APCS01], a project which aims to combine Semantic Web techniques with formal mathematics. It supports four kinds of queries, "match <pattern>" to displays the result of seeking for all statements that match the specified pattern, "instance <pattern>" to display all statements that are instances of pattern, "hint" to retrieve statements that can be applied to a given goal, and "elim" to retrieve induction principles for a given type. Whelp does not provide language features to take the theory hierarchy into account, nor does it allow the use of variable bindings from the pattern matching in a meta-level condition. The proof assistant MATITA [ACTZ07] uses the Whelp search engine in its automation realized by a tactic called "auto" to find a subset of theorems worth consideration according to the current context.

The COQ system itself provides only limited searching functionality such as the commands *Search*, *SearchAbout* and *SearchPattern* [BC04]. Similar to our approach, syntactic patterns can be specified. However, as for Whelp, meta-level conditions that use variable bindings from the selector part are not supported. Moreover, the language cannot be used within tactic expressions. Previous retrieval of lemmas was based on type isomorphisms [Del99], where the general idea was to use the type as search pattern and to carry out comparisons modulo a certain equivalence.

10.3.2 Procedural Tactics

Almost all proof systems provide constructs to perform search in the form of procedural tactics in the tradition of the *LCF* system. One major point in which their implementations differ is the handling of meta-variables and whether only a single goal or the complete context are accessible. A related discussion can be found in [ARC09]. The main focus in our work is on the intermediate language to feature the document-centric approach of mathematics, which is only supported within COQ's tactic language $\mathcal{L}_{tac}$, but has also been realized in the context of term rewriting systems, such as ELAN, APS, MAUDE, or STRATEGO.

Compared with existing work, one main feature of our approach is that the strategies can be specified within the proof document. The query language, which is a major component of our language, is non standard and gives a new dimension and perspective in the specification of tactic languages. We consider the theory environment in which the proof construction takes place as a data base and provide an explicit query mechanism to retrieve knowledge items from this environment, in particular proof operators. The retrieved knowledge items can be annotated with additional control information such as matching conditions. Building proof strategies on top of this query mechanism allows the specification of the proof operators to be used within a proof strategy based on their properties. This way, proof strategies automatically adapt to new contexts when the theory changes. Moreover, the possibility to explicitly provide a backtrack context which can individually be enriched by further backtrack conditions seems to be new. We also give a precise semantics of the language, which is not the standard in the context of tactic languages. Similarly to our approach, a non-deterministic meta-interpreter for proof methods has been given in the context of λCLAM [RSG98]. Finally, the specification of meta-level goals in an extensible language are not supported.

$\mathcal{L}_{tac}$:

Regarding intermediate tactic languages, our approach is similar to COQ's $\mathcal{L}_{tac}$ [Del02], which is an intermediate language intended to deal with small parts of proofs the user may like to automate locally. However, there is no formalization of the semantics of $\mathcal{L}_{tac}$.

$\mathcal{L}_{tac}$ introduces conveniences of higher-level programming languages to the tactic script language which are independent from the underlying programming language and is similar in spirit to our aims. More specifically, $\mathcal{L}_{tac}$ provides pattern matching against the current goal, and our syntax for sequent patterns `|-` is inspired from it. $\mathcal{L}_{tac}$ also supports to match subterms and our syntax `[t]` is also the same here, except that we also allow to impose the polarity of the subformula supposed to match by `[t]+` or `[t]-`. A real extension of our language are the means to bind results of arbitrary computations to local script variables as well as the pattern syntax with ellipsis, which probably could be included in $\mathcal{L}_{tac}$. The query mechanism, and thus the possibility to adapt to the context in which the tactic is executed, is not present in $\mathcal{L}_{tac}$. Finally, there is no language construct to trigger conditional backtracking.

APS, ELAN, MAUDE, and STRATEGO:

Several strategy languages exist for rewriting systems. The general idea is to provide a language to specify a class of derivations the user is interested in by controlling the rule applications. For example, most languages provide constructs to describe preferred application position of rewrite rules, such as bottom-up, top-down, leftmost-innermost or leftmost-outermost. Depending on the language, these are either defined by a combination of low-level primitives or build-in primitives. On a second layer, the languages provide constructs to express choice and sequencing, and recursion. Prominent examples are APS [Let93], ELAN [BK97, BKKR01], MAUDE [MOMV05], and STRATEGO [Vis01].

Similar to our work, these systems provide a declarative tactic language which is independent of the underlying implementation language of the system. Choice, sequencing and recursion operators are standard and thus comparable to our language. However, these languages stay at the term level and do not provide constructs for sequents or subgoal selection.

λClam and Isaplanner:

In [Ric02] Richardson and Smaill present a non-deterministic meta-interpreter which gives a semantics to methodical expressions in the context of the λClam system [RSG98]. Similar to tacticals, methodicals combine methods, which are abstractions from tactics and are the planning operators within proof planning [Bun88]. Their meta-interpreter works by stepwise unfolding a continuation, keeping track of the methodical expression to be evaluated. In [Ric02] the language of methodicals is extended and a nondeterministic variant presented, where choice points are pushed on a stack. Note that methodicals are not allowed to backtrack. Similarly, each reasoning state in Isaplanner [Dix05], which is a descendant of λClam, contains a continuation representing the next reasoning technique to be applied, expressing in a sense the future of the evaluation.

In contrast to this work, we use continuations exclusively to maintain backtrack points, rather than to interpret our language constructs. Moreover, we provide a language construct to enrich proof strategies with backtrack conditions. Backtracking is then controlled locally by the strategy. Only if the local strategy is not able to deal with the failure, the failure information is passed back to the invoking strategy, which in turn has to react to the failure. Moreover, as the backtrack conditions are passed down to more local strategies, solutions which would be discarded by the top-level strategy are directly discarded when being produced. Rather than interpreting the language, we rely on compilation techniques to speed up the execution time.

10.3.3 Declarative Tactics

As mentioned in the introduction, very little research has been done on the automation of the declarative style of proof. Besides the translation based approaches, which we have discussed in Section 8.5, only Isaplanner directly produces declarative proof scripts. Considering our dynamic matching constructs, a similar functionality is supported by Acl2, however, in a procedural style.

Isaplanner

Closely related to our work is Isaplanner [Dix05]. Isaplanner generates proof plans and uses Isar to represent them, that is, it also generates declarative proofs. It provides a “gap” command to represent open subgoals together with the annotation of a technique how to close such a gap. Compared to our approach, the main difference is that reasoning techniques are written as ML functions, whereas we use the underlying declarative proof language to specify the tactic. Moreover, our proof language differs from Isar by allowing meta-variables, which are not supported by Isar, despite being supported by Isabelle.

Proof Methods in Ωmega

In the previous version of Ωmega, so-called *proof methods* were declaratively represented by *proof schemas*. Proof schemas were partial proofs in a natural deduction style (see [HKC98]). In contrast to our approach, methods were implemented directly in the programming language, no declarative proof language was used. Moreover, there was no possibility to pass control information in the form of a continuation.

ACL2

The matching part in case constructs of our tactic language is related to the extended meta-functions in ACL2 [JKK$^+$05] which allow to access the current goal clause. The ACL2 meta-functions need to be proved correct in order to be usable by the ACL2 reasoner. From an LCF point of view, this is a way to include derived reasoning steps in the kernel proof rules, thus extending the kernel rules. In contrast to this our approach remains entirely in the LCF tradition since the declarative strategies generate proof scripts, which still need to be evaluated by the underlying (LCF-based) proof script interpreter. The possibility to perform arbitrary computations and bind the results to a term pattern, like the call to `maxima-factor` in the strategy ***factorbound*** is close to ACL2's `bind-free`, which takes an arbitrary binding list and adds it to the local context. This is also possible with our pattern approach by writing `X_1 .. X_N`, which has the advantage that the names of the local variables can be specified by the writer of the strategy. It would be possible to accommodate the `bind-free`-style in the pattern syntax, but so far we have not encountered situations where this was required. Moreover, the examples presented in [JKK$^+$05] also bind only one variable.

Mixed Initiative/Advisable Planning

Mixed initiative planning systems are systems in which humans and machines collaborate in the development and management of plans. The idea is that by synthesizing the strength of both men and machine to build plans more quickly, or more reliable plans, or to be able to create plans which could not be found without the interaction of a human user. Most systems support interaction for certain low-level operations, such as ordering goals for expansion, selecting operators to apply, choosing instantiations for planning variables [DT95, Wil93]. However, only an interaction at a higher abstraction level really allows the human to efficiently cooperate with the planner. Sometimes, this style is called *advisable planning* [Mye96], because the human gives only hints or sketches of how to solve a given problem. Our approach falls into this category, as underspecified proof sketches can be seen as an advice to the underlying proving system. By using declarative tactics, this knowledge can even be stored and be made applicable in an automated setting. We refrain from giving a detailed comparison with advisable planning systems here, as we share only similar ideas at an abstract level.

10.4 Summary

In this chapter we introduced a strategy language to express generic and specialized reasoning procedures. Our language is independent of the underlying programming language and allows the specification of proof strategies within the proof document (Contribution A2(ii), Section 1.1), thus featuring the document centric approach which is not supported by most proof assistants today. Moreover, it allows the user to express reasoning patterns both in a bottom-up style as known from classical tactics, but also in a top-down style as known from proof planning. Rich features to describe and classify proof states in a declarative way as well as language constructs to control the search space including their precise semantics have been defined. Finally, we introduced the new notion of a declarative strategies, which takes advantage of the close correspondence of declarative proof scripts and proof plans and supports the automatic generation of declarative proof scripts.

Part IV

Applications

11 Using Assertion Level for Tutoring

In this section, we describe the integration of ΩMEGA within a tutor system to reconstruct and verify proof steps uttered by students, as studied within the DIALOG project [BFG$^+$03, BHKK$^+$07a]. Thereby, we address the problem of how to deal with ambiguous and underspecified proof steps, such as partial subgoals, which typically occur within this scenario. Moreover, we sketch how our approach can be extended to detect and analyze student errors and how to generate hints that can be given to the student on demand.

11.1 Motivation and Context

Computer-based learning systems are increasingly popular since they allow for independent learning and individualized instruction. The research and development of *intelligent tutoring systems (ITS)* is a rapidly growing area and these systems are huge integrated tools that combine domain specific knowledge, pedagogical methods, student models as well as intelligent, user-adaptive tutoring components. However, although there exist strong tools for teaching mathematics and to a certain extent mathematical computations, there are hardly any training modules for theorem proving in these tools. Only a few attempts have been made to build dynamic proof tutoring systems.

The reason is that unlike for other kinds of mathematical exercises it is not sufficient for an ITS to provide feedback only on the *solution* of a problem, i.e., on a full proof. Rather, the ITS must provide feedback on intermediate stages of the proof, but in contrast to other teaching domains, it is common for a proof problem to have different solutions. Consequently, the standard ITS technique of pre-authoring solutions (including problem- and situation-specific hints) are unsuited – if not unfeasible. Rather, there is a need for dynamic techniques that allow the assessment of the student's proof steps on a case-by-case basis and to generate the appropriate feedback. For tutoring theorem proving, the feedback can take the form of confirming correct steps, drawing the student's attention to errors, and offering domain specific hints when the student gets stuck. Thus the envisioned scenario for tutoring theorem proving is that the student is free to build any valid proof of the theorem at hand, and each proof step is analyzed in the context of the partial proof constructed so far. In case the tutor is asked to give a hint, the hint is generated in the context of the current proof and is therefore exactly tailored to the situation in which the hint was requested. Note that this approach is compatible with the so-called *buggy rule approach* [BV80], in which typical errors are modeled by so-called buggy rules which also detect typical student errors.

The framework in which our research has been carried out is the DIALOG project [BFG+03, BHKK+07a], which had the final goal of natural tutorial dialog between a student and a mathematical assistance system. In our scenario, we suppose a student is taking an interactive course in some mathematical field within a web-based learning environment, such as ACTIVEMATH [MS05], the system developed at the DFKI by Erica Melis and her research group. ACTIVEMATH is a generic web-based learning system that dynamically generates interactive (mathematical) courses adapted to the student's goals, preferences, capabilities, and knowledge. It comes with several desirable features such as: user modeling, user-adapted content selection, sequencing, and presentation; support of active and explorative learning by external tools; use of (mathematical) problem solving methods, and reusability of the encoded content as well as interoperability between systems, which can be accessed within the architecture of DIALOG. The ACTIVEMATH learning environment is equipped with user modeling and monitoring facilities and maintains a dynamically updated student model (SM) containing information about the axioms, definitions, theorems (hence the assertions) and the proof techniques the student has studied and mastered so far. We also assume an idealized student model (ISM) set up by the author of each learning unit, which specifies the mathematical material that a student ideally should know after studying the unit.

In the course of a tutorial session in DIALOG, the student's task is to build a proof by performing natural language utterances which may contain proof steps. These are dynamically evaluated in order to generate feedback and hints. The typical workflow of such a dialog move is depicted in Figure 11.1. The student's input, which may contain natural language, is analyzed to get a formal representation of the proof step. This step in turn is given to the domain reasoner, which tries to reconstruct and thus verify it. The reconstruction is further analyzed, and a didactic teaching module decides on and generates feedback in an abstract form. Finally, this abstract feedback is translated into natural language and shown to the student.

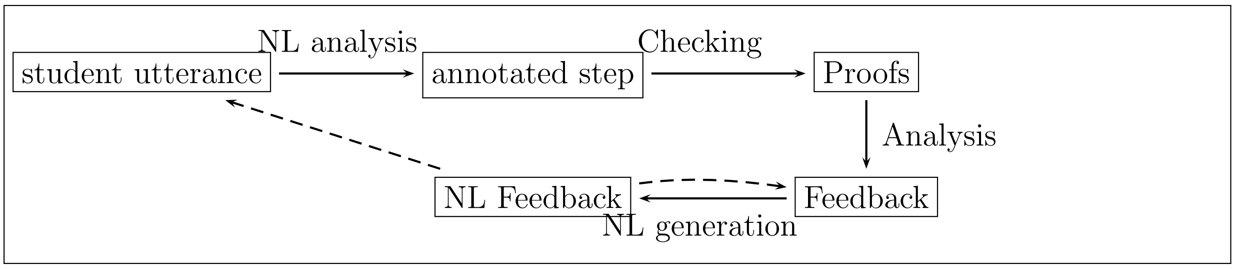


Figure 11.1: Workflow of a dialog move

In the following, we focus on the mathematical domain reasoning. The analysis of the student's natural language utterances itself is a very challenging area which we will not consider here (see [BHKK+07b] for details on the problem). Instead, we assume the existence of an NLP module that can deliver the formula that the student intended to utter, together with a proof step type containing information about the context of the formula, such as whether the formula is intended to represent a new goal or fact. For possible approaches to such an analysis we refer to [HW05, Wag10, WBH+09]. Note that this does not necessarily mean that the obtained information is unambiguous and complete. The more likely and, from our point of view, more interesting case is that the proof step statement is incomplete, in the sense that information necessary to construct a formal proof has been left out. This comprises not only information about how many or what steps were employed, or which of the assumptions or derived formulas were used, but also relevant subgoals which the student wants to tackle later on and which are not mentioned. From a formal point of view, the latter are the most difficult to handle, as they make a statement logically incorrect and therefore need a special treatment.

Note that an incomplete proof step is not necessarily a faulty proof step: when writing proofs, humans typically omit information they consider unimportant or trivial. Simply noting that a proof step is false or incomplete is an inadequate basis to generate useful hints for the student. Moreover, it is desirable to provide feedback for the step immediately. It has been shown that immediate feedback has a positive effect on the performance of the student [HT07, MRM+95]. Anderson observes a declined learning rate as a consequence of deferred feedback [ABB93].

11.2 A Corpus of Mathematical Tutorial Dialogs

Our approach to the verification of proof steps is motivated by phenomena found in two corpora of tutorial dialogs, collected in two studies between students and experienced mathematics teachers following the Wizard-of-Oz paradigm [Kel83]. The goal of the experiments was to collect data on the use of natural and mathematical language in a tutorial interaction and on the behavior of students and tutors with respect to the theorem proving task. While the first experiment focused on linguistic aspects, the second experiment investigated mathematical domain reasoning tasks and linguistic phenomena in tutorial dialogs. Due to its focus on domain reasoning, its more comprehensive corpus and its higher actuality, we use the second experiment as a basis for the evaluation of our algorithm.

11.2.1 Corpus of the Second Experiment

After having seen some preparatory material introducing the theory of binary relations, students were asked to solve four exercises in a session with the tutorial system. The collected corpus contains the data of 37 subjects. The thirty-seven experiment sessions include a total of 1917 dialog turns (980 by the wizards and 937 by the students). The students tried maximally four different exercises each. However, since the duration of the experiment session was limited to two hours, some students did not have the opportunity to do all exercises. The fourth exercise was considered a "challenge exercise", and therefore it was expected that only some students would attempt it. On average, each student attempted 2.7 exercises (i.e., we have collected a total of 100 exercise-subdialogs). Tutors rated each proof step with respect to correctness, granularity (or proof step size) and relevance to the current task.

Figure 11.2 shows several fragments of tutorial sessions in which the student has been instructed to prove the theorem $(R \circ S)^{-1} = (S^{-1} \circ R^{-1})$, where R and S are relations. In the figure, **S** refers to a student turn and **T** to a tutor turn.

The approach taken by the student in the first example (Figure 11.2(a)) is to apply set extensionality and then to show that the subset relation holds in both directions. The student begins in utterance **S1** by directly introducing a pair (x, y) in the set $(R \circ S)^{-1}$. This is rated as correct by the tutor, who recognizes that the student wants to prove

S1: *Let* $(x, y) \in (R \circ S)^{-1}$.
T1: *Good.* $\boxed{\textit{correct}}$
S2: *It follows that* $(y, x) \in (R \circ S)$.
T2: *That's correct.* $\boxed{\textit{correct}}$
S3: *Therefore* $\exists z \in M : (y, z) \in R \wedge (z, x) \in S$.
T3: *Yes, that's right.* $\boxed{\textit{correct}}$
S4: *This implies:* $\exists z \in M : (z, y) \in R^{-1} \wedge (x, z) \in S^{-1}$.
T4: *Your deduction is correct. You are on a good way.* $\boxed{\textit{correct}}$
S5: *It follows:* $(x, y) \in S^{-1} \circ R^{-1}$.
T5: *Yes. Therefore it is shown that every element from* $(R \circ S)^{-1}$ *already lies in* $S^{-1} \circ R^{-1}$. $\boxed{\textit{correct}}$

S6: *Now let* $(x, y) \in S^{-1} \circ R^{-1}$.
T6: *O.k.* $\boxed{\textit{correct}}$
S7: $\Rightarrow \exists z \in M : (x, z) \in S^{-1} \wedge (z, y) \in R^{-1}$.
T7: *Yes.* $\boxed{\textit{correct}}$
S8: $\Rightarrow \exists z \in M : (z, x) \in S \wedge (y, z) \in R$.
T8: *This deduction is also correct.* $\boxed{\textit{correct}}$
S9: $\Rightarrow (y, x) \in R \circ S$.
T9: *This deduction is again correct.* $\boxed{\textit{correct}}$
S10: $\Rightarrow (x, y) \in (S \circ R)^{-1}$.
T10: *Congratulations! With this you have shown both inclusions. Your solution is now complete.* $\boxed{\textit{correct}}$

(a) One sample proof dialog for the exercise $(R \circ S)^{-1} = S^{-1} \circ R^{-1}$ from the 2005 Wizard-of-Oz experiments (reproduced from [BDSA07]).

S8: *let* $(x, y) \in (R \circ S)^{-1}$
T9: *correct*
S10: *hence* $(y, x) \in (S \circ R)$
T11: *incorrect*

(b) Example of a faulty proof step

S8a: *we consider the subgoals* $(R \circ S)^{-1} \subset S^{-1} \circ R^{-1}$ *and* $(R \circ S)^{-1} \supset S^{-1} \circ R^{-1}$

S8b: *first, we consider the subgoal* $(R \circ S)^{-1} \subset S^{-1} \circ R^{-1}$

(c) Example of backward steps

Figure 11.2: Examples of tutorial dialogs from the corpus

both directions separately and that the introduction of the pair (x, y) is useful due to the definition of subset. The student continues the proof and finally solves the exercise.

Figure 11.2(b) shows a variant of the proof approach shown in (Figure 11.2(a)) containing a faulty step. Whereas the first step is the same as in the previous proof, the student interchanges S and R in the second step.

Two alternative ways that the student could have started the same exercise are shown on the right in Figure 11.2(c). In **S8a** the student explicitly splits the proof into two subgoals with an application of set extensionality. In **S8b** the same rule is applied, but only one of the two resulting subgoals is explicitly presented.

11.2.2 Phenomena Observed in the Corpus

We analyzed those utterances from the corpus which contain contributions to the theorem proving task. We were able to identify a number of general phenomena which must be accounted for in order to correctly verify (or reject) the proof steps that students perform and to maintain correct consistent representations of the proofs they are building.

Underspecification Some subset of the complete description of a proof step is often left unstated. Utterance **S8b** in Figure 11.2(c) is an example of a number of different types of this underspecification which appear throughout the corpus. The proof step in **S8b** includes the application of set extensionality, but the rule is not stated explicitly, nor are the arising subgoals specified. The student also does not specify which of the resulting subgoals he is now proving and the existence of the other subgoal. Further, the number of steps needed to reach this proof state is not given. Part of the task of analyzing such steps is to instantiate the missing information so that the formal proof object is complete.

Incomplete Information Proof steps can, in addition to issues of underspecification, be missing information which is necessary for their verification by formal means. For instance the utterance **S8** in Figure 11.2(b) is clearly a contribution to the proof, but since the step only introduces a new variable binding, there is no assertion whose truth value can be checked. However, anticipating that the student is using the definition of subset $A \subset B \Leftrightarrow x \in A \Rightarrow x \in B$ allows us to determine that the new variable binding is indeed useful. Utterance **S8b** is also a correct contribution, but the second subgoal is not stated. However, this second subgoal is necessary to verify that proving the subset relation is part of justifying the equality of the sets, since one subgoal alone does not imply the set equality which is to be shown.

These examples are evidence that verification in this scenario is not simply a matter of logical correctness, but must take into account for instance proof context.

Ambiguity Ambiguity pervades all levels of the analysis of the natural language and mathematical expressions that students use. Even in fully specified proof steps an element of ambiguity can remain. For example in any proof step which follows **S8a**, we cannot know which subgoal the student has decided to work on. Also, when students state formulas without natural language expressions, such as "hence" or "conjecture", it is not clear whether the formula is a newly derived fact or a newly introduced conjecture. Again, this type of ambiguity can only be resolved in the context of the current proof, and when resolution is not possible, it does not mean that the proof step is incorrect, but the ambiguity must be propagated. This must be done by maintaining multiple parallel interpretations, which are retained until enough information has been provided by the proof context to decide whether they are still consistent. Note that we do not argue that the tutor should accept ambiguous proof steps in general, but that the ability to identify how such a step fits into the current proof context is certainly beneficial in order to produce feedback.

11.2.3 Proof Step Types and Interface

The corpus shows that proof steps are embedded in utterances which carry information about the type of the proof step which is important for its verification, such as the phrase "we consider the subgoal" in utterance **S8b** in Figure 11.2(c). With the background in the field of declarative proof script languages it is a natural choice to embed a single utterance in such a language and to use it as a vehicle to interchange information between the NL analysis module and the domain reasoner. Note that proof languages already provide the possibility to indicate which assertions are to be used within the verification as well as the places to which the assertions are applied to. However, while we can make use of representations developed within the field of interactive theorem proving, the processing of a proof step within the setting of tutorial dialog differs from the processing in a pure verification setting with respect to the following points:

- In a pure verification setting, it is sufficient to verify a step. The verification itself is usually not of interest and needs not to be further processed. In contrast, in a tutorial setting we need to consider several, if not all, possible proofs of the given step (and relate them to the knowledge of the student) and need to further analyze the verification.
- In a pure verification setting, we can assume the user to be an expert in the problem domain as well as in the field of formal reasoning. This has several influences on

the processing model: (i) inputs can be expected to be correct and just need to be checked, (ii) steps can lazily be verified until a (sub)proof is completed, (iii) feedback is limited to checkable and not checkable.

In contrast, in a tutorial setting, the user is neither a domain expert nor an expert in formal reasoning. The underlying mechanisms need to be hidden from the user, direct and comprehensive feedback has to be provided at each step. Therefore, it is for example a requirement to anticipate why the assumption is made, in contrast to a lazy checking once the conclusion has been obtained.

- In a pure verification setting, we can assume the user to indicate when the proof of a subgoal is finished (as usually done by so-called *proof step markers* in the proof language). However, in the tutorial setting this information is implicit. Similarly, we must be able to perform backward steps where some of the new proof obligations have not yet been shown.

The types of proof step we consider are **let**, which introduces new variables, **hence**, which indicates a forward reasoning step, **subgoal**, which indicates a reduction of a goal to subgoals, **assume** to make an assumption necessary to conduct a proof by contradiction, and **conjecture** to state a lemma. We additionally use **trivial** to claim that a proof goal has been closed and **back** to undo a proof step. Using these steps, we can model the example dialog shown in Figure 11.2(a) as shown in Figure 11.3.

s1: **let** $(x, y) \in (R \circ S)^{-1}$
s2: **hence** $(y, x) \in (R \circ S)$
s3: **hence** $\exists z \in M : (y, z) \in R \wedge (z, x) \in S$
s4: **hence** $\exists z \in M : (z, y) \in R^{-1} \wedge (x, z) \in S^{-1}$
s5: **hence** $(x, y) \in S^{-1} \circ R^{-1}$
s6: **let** $(x, y) \in S^{-1} \circ R^{-1}$
s7: **hence** $\exists z \in M : (x, z) \in S^{-1} \wedge (z, y) \in R^{-1}$
s8: **hence** $\exists z \in M : (z, x) \in S \wedge (y, z) \in R$
s9: **hence** $(y, x) \in R \circ S$
s10: **hence** $(x, y) \in (S \circ R)^{-1}$

Figure 11.3: Sequence of proof step input for the ΩMEGA-Tutor generated from the above example dialog.

11.3 Mental Proof States

After having fixed the representation for the input of a students utterance, the next step in the workflow consists of verifying/rejecting the utterance of the student, based on the previous development of the proof. Therefore, the possible states the student might be in need to be represented and maintained. We call such a state *mental proof state* (MPS).

Initially, the MPS is unique and consists of the exercise given to the student. Given a proof step s, the general idea of the reconstruction algorithm is to determine all possible successor states which are consistent with the utterance s. Let us stress again that due to ambiguity and underspecification several consistent successor states are possible (as in the case of statement **S8b** shown in Figure 11.2(c)). Therefore, the verification algorithm works on a list of MPS rather than on a single one.

11.3.1 Representing the Possible MPS

As the $\mathcal{PDS}$ does already provide mechanisms to simultaneously represent several proof attempts in parallel, it is natural to also use it to represent all of the possible MPS the student might be in. More specifically, each agenda (see Chapter 6) determines one possible MPS. Given a problem and a theory, the inferences are automatically constructed from the formulas $\mathcal{I}$ and the *initial* $\mathcal{PDS}$ is constructed. The initial $\mathcal{PDS}$ consists of one initial task $T = \Gamma \vdash \Delta$ where Γ contains the assumptions of the problem and Δ is the proof problem to be shown. The *initial MPS* is determined by the *initial agenda*, containing only the initial task: $\mathcal{A} = \langle \underline{T}; \emptyset \rangle$. The set of all consistent MPS is denoted by $\mathfrak{A}$.

Before explaining the checking algorithm in detail, we show the formalization of the underlying theory in Figure 11.4.

theory *woz2;*
newtype *relation;*
newtype *object;*
define $\in$*:object* $\times$ *relation* $\rightarrow o$*;*
define $(\_,\_)$*:object* $\times$ *object* $\rightarrow$ *object;*
define $\subset$*:relation* $\times$ *relation* $\rightarrow o$*;*
    $\forall R, S.(R \subset S) \Leftrightarrow \forall x, y.(x, y) \in R \Rightarrow (x, y) \in S);$
define $\cap$*:relation* $\times$ *relation* $\rightarrow$ *relation;*
    $\forall R, S, x, y.(x, y) \in (R \cap S) \Leftrightarrow ((x, y) \in R \wedge (x, y) \in S);$
define $\cup$*:relation* $\times$ *relation* $\rightarrow$ *relation;*
    $\forall R, S, x, y.((x, y) \in (R \cup S)) \Leftrightarrow ((x, y) \in R \vee (x, y) \in S);$
define $^{-1}$*:relation* $\rightarrow$ *relation;*
    $\forall R, x, y.((x, y) \in (R^{-1})) \Leftrightarrow (y, x) \in R);$
define $\circ$*:relation* $\times$ *relation* $\rightarrow$ *relation;*
    $\forall R, S, x, y.((x, y) \in R \circ S) \Leftrightarrow \exists z.((x, z) \in R \wedge (z, y) \in S);$
define *trans:relation* $\rightarrow o$*;*
    $\forall R.trans(R) \Leftrightarrow \forall x, y, z.(((x, y) \in R) \wedge ((y, z) \in R)) \Rightarrow ((x, z) \in R);$
axiom *"setr="* $\forall A :$ *relation*$, B :$ *relation*$.A = B \Leftrightarrow ((A \subset B) \wedge (B \subset A));$
conjecture *"distributivity of composition over union"*
    $\forall R, S, T.((R \cup S) \circ T) = (R \circ T) \cup (S \circ T);$
conjecture *"exercise w"* $\forall R, S.(R \circ S)^{-1} = S^{-1} \circ R^{-1};$
conjecture *"velleman4-p179-3"* $\forall R.trans(R) \Leftrightarrow R \circ R \subset R;$

Figure 11.4: Formalization of the background theory

11.3.2 Updating the MPS

Given a set of possible MPS $\mathfrak{A}$ and a preprocessed utterance s, we must determine all possible successor states which are consistent with the utterance s. For the sake of simplicity we only show how to determine the successor states of a single MPS $\mathcal{A} \in \mathfrak{A}$, denoted by $\varphi(\mathcal{A})$. Combining the result for each $\mathcal{A} \in \mathfrak{A}$ gives the complete set of successor states. If no agenda can provide a consistent successor state, i.e.,$\cup_{\mathcal{A} \in \mathfrak{A}} \varphi(\mathcal{A}) = \emptyset$, the step is classified to be *incorrect*.

Given a proof step s to be checked and an agenda $\mathcal{A} \in \mathfrak{A}$, $\varphi(\mathcal{A})$ is determined in four steps: (i) A depth limited BFS search is performed, in which the current task of the agenda, denoted by **current** $(\mathcal{A})$, is expanded[1] (including some control knowledge to

[1]Note that without restriction we can assume that **current** $(\mathcal{A}) \neq \perp$ in (i) (otherwise we transform

```
strategy tutor-let
repeat
  with depthlimit=5, mode=bfs, level
  using
    select * from current as backward |-*
until [fact]^-{nopobs},*|-*
  where (every #'(lambda (var) (eigenvar-is var)) (freevars fact))
```

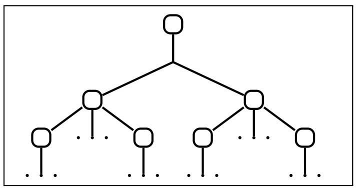
(a) Checking of **let** steps

```
strategy tutor-hence
repeat
  with depthlimit=5, mode=bfs, level
  using
    select * from current as forward *|-
until [fact]^-{nopobs},*|-*
```

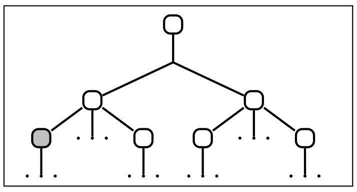
(b) Checking of **hence** steps

Figure 11.5: Tutor strategies for **hence** and **let**

cut off irrelevant branches of the search space), resulting in a set of successor nodes, (ii) from this, consistent successor nodes are selected and partial agendas are created from them, (iii) the partial agendas are completed, and finally (iv) the $\mathcal{PDS}$ is cleaned. The overall result is a confirmation of whether the step could be verified, along with the side-effect that the $\mathcal{PDS}$ has been updated to contain exactly the possible MPS resulting from performing the step. We now illustrate each step of the algorithm schematically.



Step (i) The node **current** ($\mathcal{A}$) is expanded using a depth limited BFS search. The depth limiter specifies how many assertion steps[2] the student is allowed to perform implicitly. Intuitively we can think of this bound as reflecting the experience of the student, consequently it should be based on a student model. Another possibility is to have a fixed bound and to express the experience of the student only by available facts in the model. The bound is needed to guarantee termination of the verification algorithm, which might otherwise not terminate.



Step (ii) Whereas step (i), (iii), and (iv) are independent of the proof step type c, step (ii) differs for each c. Therefore we introduce a filter function Θ_c to filter consistent successor states for each type c. We apply this filter function Θ_c to get those successor states which possibly represent the student's MPS after applying the utterance. Note that in general there may be several successor states satisfying Θ_c. To further restrict the consistent successor nodes only those with minimal distance to the expansion node are stored. The filters are expressed within the strategy language as solution conditions of the **repeat** construct. For the types introduced in Section 11.2.3 we define the following filters:

the $\mathcal{A}$ into a set of agendas representing all possibilities to select a task in $\mathcal{A}$).

[2]Note that the correspondence of student steps to calculus steps may vary for each calculus.

let A node is classified by Θ_{let} to be consistent if its corresponding task contains f as a subformula with negative polarity, no conditions and the free variables of f have been introduced. A negative subformula f of a formula F has no conditions if the decomposition of F to obtain f in the sequent calculus introduces no branching and f occurs on the left-hand side of the resulting sequent. The corresponding strategy is shown in Figure 11.5(a).

hence A node is classified by Θ_{hence} to be consistent if its corresponding task contains the introduced fact as a subformula with negative polarity and no conditions. The search is restricted in the sense that only new facts are derived during the expansion of the agenda. The corresponding strategy is shown in Figure 11.5(b).

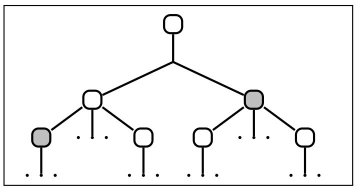
subgoals Given the input **subgoals** $s_1 \dots s_n$ the subgoal filter tries to find successor nodes $s_1, \dots, s_n$ in different branches such that each s_i occurs as a positive subformula in s_i. During the expansion, only reduction steps are allowed, i.e., steps that reduce the current goal. The corresponding strategy is shown in Figure 11.6(b).

assume Assume works similarly to let, additionally it checks for proof by contradiction (positive subformula $\neg f$) or contrapositive (positive subformula f). The corresponding strategy is shown in Figure 11.6(a).

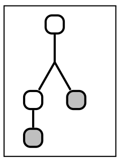
conjecture Conjecture starts a new proof tree and tries to prove the conjecture using the currently available knowledge. If the conjecture could be proved, it is inserted into the knowledge base and hence available for the main proof.

done Done is used to indicate that the proof is completed. It checks whether the current goal(s) can be closed. The corresponding strategy is shown in Figure 11.7.

Step (iii) The agendas obtained from step (ii) can be partial, i.e., not all subgoals that must be solved are in the agenda. Such situations occur if the prover reduces a task to more than one subtask, but the Θ_c does not select a node from each branch. In this case, it is reasonable to assume that the user will prove the arising subtasks later. In particular, he has not performed any actions on the missing subtasks. Hence, we extend the partial agenda by the tasks introduced by the reduction. We extend the agenda as shown on the right. An example for this situation is step **S8b** in Figure 11.2: The student specifies only one subgoal $(R \circ S)^{-1} \subset S^{-1} \circ R^{-1}$, but to be logically correct, the subgoal $S^{-1} \circ R^{-1} \subset (R \circ S)^{-1}$ must also be proved. It is the latter subgoal that is collected and inserted in the agenda.



Step (iv) Usually there will be many nodes which are generated by the search but are rejected by the filter. All these nodes are removed from the $\mathcal{PDS}$, resulting in the proof state shown on the right.



```
strategy tutor-assume
case ass of ¬φ ->
  repeat
    with depthlimit=5, mode=bfs, level
    using
      select * from current as backward */-
  until (or "[¬φ]^-{nopobs}"
            "[φ]^+,*/-*")
            "[¬φ]^+,*/-*")
default ->
  repeat
    with depthlimit=5, mode=bfs, level
    using
      select * from current as backward */-
  until (or "[ass]^-{nopobs}"
            "[ass]^+,*/-*")
```

(a) Checking of **assume** steps

```
strategy tutor-subgoal
repeat
  with depthlimit=5, mode=bfs, level, partial
       task-solved=[subgoal]^+,*/-*
         where (member subgoal subgoals)
  using
    select * from current as backward */-
until (every #'(lambda (subgoal) (exists-solution subgoal solutions))
      subgoals)
```

(b) Checking of **subgoal** steps

Figure 11.6: Checking strategies for tutoring

11.4 Example Verification

In order to illustrate how the verification algorithm works, we will step through the verification of utterance **S1** from Figure 11.2(a), beginning with the initial MPS and finishing with the MPS extended by the proof step. The initial MPS is $\{\langle \underline{\vdash (R \circ S)^{-1} = S^{-1} \circ R^{-1}}; \emptyset\rangle\}$ and the proof step to be verified is **let** $(x, y) \in (R \circ S)^{-1}$.

Having expanded the current task (step (i), shown in Figure 11.8), we apply the filter Θ_{let} to find the set of newly-created tasks which are consistent with the given proof step. Of the tasks in the tree, only the node containing the task T_k passes, since the formula in the proof step appears on the left-hand side of the sequent. Now that we have found the consistent successor tasks, we must complete all partial agendas (step (iii)). Because the decomposition of the task T_0 introduced a subgoal split, the task T_j must be proved in addition to T_k. The resulting agenda is therefore $\{\langle \underline{T_k}, T_j; \emptyset\rangle\}$, that is, T_k is now the current task, and T_j is still to be proved. Finally in step (iv), we prune the nodes which were rejected by the filter, resulting in the proof state shown in Figure 11.9. This becomes the initial state for the verification of the student's next proof step.

This example has illustrated the verification algorithm for a single agenda with a single successor state. For ambiguous steps, that is steps which result in more than one successor task after the filter in step (ii), we simply create a new agenda for each successor, and maintain these in the set of agendas $\mathfrak{A}$. Step (iii) is done separately for each new agenda,

```
strategy tutor-done
repeat
  with depthlimit=5, mode=bfs, level
  using
    select * from current as close
    union
    select standard-simplify from base
until (is-solved agenda)
```

Figure 11.7: Checking of **done**

and only those nodes which do not occur in any agenda are pruned in step (iv). An example of such an ambiguity is the verification of proof step **S8a**. Here two successor agendas are created, $\{\langle \underline{T_i}, T_j; \emptyset\rangle\}$ and $\{\langle T_i, \underline{T_j}; \emptyset\rangle\}$, differing only in which of the two tasks is the current task.

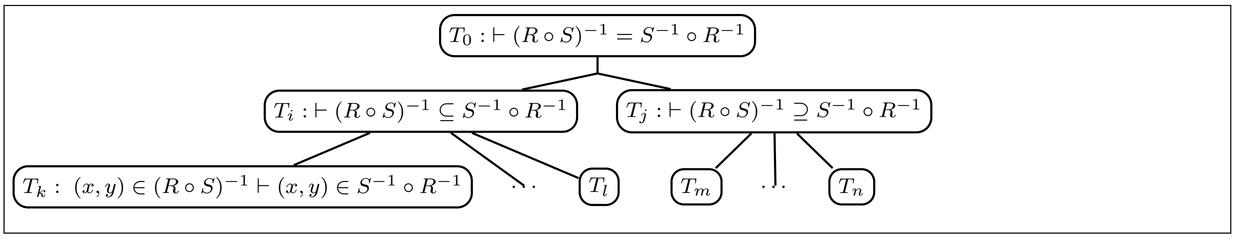
For the case in which the current proof state already contains multiple agendas, the entire verification process is carried out for each in turn. This is the point at which ambiguities introduced in previous steps can be resolved: those agendas which do not lead to successor states are deleted, and those which have successors are maintained. For example, each proof step following **S8a** will belong to either the left or right branch of the proof only, and will have no successor tasks in the other branch. Thus the agenda in which the "wrong" current task had been chosen will be deleted.

11.4.1 A Note on the Search

To verify a step s uttered by the student a heuristic search is performed. Thereby, all MPS are expanded, expecting to find a proof state that is consistent with the utterance s. In the context of AI planning, one often differs between forward and backward planning. Forward planning corresponds to an algorithm that expands the initial state (and subsequently derived states) until one of the derived states contains the goal. In contrast, backward planning starts with the goal state, which is usually only partially specified, and reduces it to a set of new goal states by finding planning operators that achieve the goal and trying to satisfy their preconditions.

With respect to this terminology, our search is a forward search in which the actions are inferences and the states are agendas. We rely on a forward search due to the following reasons:

- A given utterance s contains only very little information about the concrete shape of the goal state. Given for example the statement **let** s, the only fact that is known is that s occurs as an assumption in one of the successor tasks. Backward search would correspond to the process of considering all inferences that introduce an assumption



$T_0 : \vdash (R \circ S)^{-1} = S^{-1} \circ R^{-1}$
$T_i : \vdash (R \circ S)^{-1} \subseteq S^{-1} \circ R^{-1}$
$T_j : \vdash (R \circ S)^{-1} \supseteq S^{-1} \circ R^{-1}$
$T_k : (x,y) \in (R \circ S)^{-1} \vdash (x,y) \in S^{-1} \circ R^{-1}$
T_l
T_m
T_n

Figure 11.8: The expanded task after step (i) of verification (abbreviated).

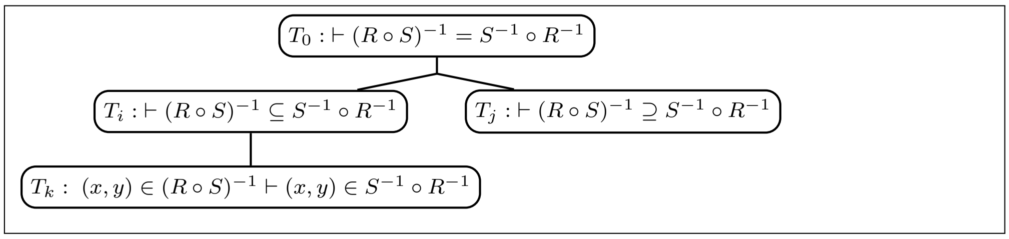

$T_0 : \vdash (R \circ S)^{-1} = S^{-1} \circ R^{-1}$
$T_i : \vdash (R \circ S)^{-1} \subseteq S^{-1} \circ R^{-1}$
$T_j : \vdash (R \circ S)^{-1} \supseteq S^{-1} \circ R^{-1}$
$T_k : (x,y) \in (R \circ S)^{-1} \vdash (x,y) \in S^{-1} \circ R^{-1}$

Figure 11.9: The resulting proof state after verification.

that unifies with s. As the corresponding inference substitution can only partially be derived, this approach introduces unknowns in the form of meta-variables which are difficult to handle in a backward search and make many inferences applicable. By relying on a forward search, this problem can be avoided.

- During the search, an invariant is that an agenda always represents a valid proof state. By expanding a given proof state only by valid actions, it is guaranteed that only reachable and consistent proof states are generated.

- The search algorithm can easily be formulated, and the resulting $\mathcal{PDS}$ can easily be analyzed by subsequent modules (such as granularity and relevance).

- Faulty proof steps would lead to spurious states, for which we do not expect correct inferences to be applicable.

There is the preconception that forward search is less efficient than backward search. Note however that from a theoretical point of view the source of inefficiency is similar: While a forward search might generate states which do not reach the goal, a backward search might introduce states that cannot be reached from the initial state. Moreover, if there is only very little information about the goal state, this approach might not be applicable, as it is the case within our setting. Even in the context of classical planning, forward state-space search has become a popular approach in AI planning within years. This is due to the development of domain independent heuristics, leading to state-of-the-art performance in many problem domains, including the classical Strips domain [Hof01], optimal planning [HHH07], temporal metric planning [MBD03], nondeterministic planning [BKS06] to name a few.

Note further that even though our approach is based on forward search, it allows for a variety of further optimizations, which have not been taken into account in the current implementation:

- In the current implementation, there is no *cycle checking* or multiple-path pruning. Cycle checking means that a path can be pruned if a state is generated which is already on the path of the node. While we do not expect cycles when working backwards from the goal, they are naturally generated when applying inferences in forward direction (**hence** steps). *Multiple-path pruning* generalizes and subsumes a cycle check. It allows the pruning of a new state if it has already been found before.

- The branching factor could further be refined by heuristics. For example, a reasonable assumption is that new Eigenvariables are only explicitly introduced by **let** statements. Thus, we could restrict the actions during forward exploration to those not introducing new variables. Note that such a heuristic would still be domain independent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(x,y) \in S^{-1} \circ R^{-1} \vdash \text{"—"}}\,Ax}{(z,y) \in R^{-1} \wedge (x,z) \in S^{-1} \vdash \text{"—"}}\,Def\circ}{(y,z) \in R \wedge (x,z) \in S^{-1} \vdash \text{"—"}}\,Def^{-1}}{(y,z) \in R \wedge (z,x) \in S \vdash \text{"—"}}\,Def^{-1}}{(y,x) \in (R \circ S) \vdash \text{"—"}}\,Def\circ}{(x,y) \in (R \circ S)^{-1} \vdash (x,y) \in S^{-1} \circ R^{-1}}\,Def^{-1}}{(R \circ S)^{-1} \subset S^{-1} \circ R^{-1}}\,Def\subset \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(x,y) \in (R \circ S)^{-1} \vdash \text{"—"}}\,Ax}{(y,x) \in (R \circ S) \vdash \text{"—"}}\,Def^{-1}}{(z,x) \in S \wedge (y,z) \in R \vdash \text{"—"}}\,Def\circ}{(x,z) \in S^{-1} \wedge (y,z) \in R \vdash \text{"—"}}\,Def^{-1}}{(x,z) \in S^{-1} \wedge (z,y) \in R^{-1} \vdash \text{"—"}}\,Def^{-1}}{(x,y) \in S^{-1} \circ R^{-1} \vdash (x,y) \in (R \circ S)^{-1}}\,Def\circ}{S^{-1} \circ R^{-1} \subset (R \circ S)^{-1}}\,Def\subset}{\vdash (R \circ S)^{-1} = S^{-1} \circ R^{-1}}\,Def=$$

Figure 11.10: Annotated ΩMEGA assertion level proof for the example dialog.

- It seems natural to integrate domain dependent (but still complete) heuristics on demand. For example, it appears reasonable to analyze the structure of the given formula and to choose an appropriate specialized checking strategy accordingly.

- The structure of the generated nodes is not yet taken into account. A possible extension is to make use of reachability heuristics and to estimate the costs to the goal, e.g., by making use of symbol distances, or to perform a more sophisticated difference analysis such as rippling.

- At each step, we can divide the set of available inferences into two categories: Those that can possibly introduce the desired fact or goal and those which cannot. The former can be determined by a static analysis, as they must unify/match with the fact or the subgoal. Such an analysis could lead to a two step exploration, only applying the inferences of the second category if no consistent state could be generated by the inferences of the first category.

11.5 Evaluation

We have evaluated our verification module with 17 tutorial dialogs taken from the Wizard-of-Oz corpus described in Section 11.2.1, containing a total of 144 proof steps. The steps within a single dialog are passed to the verification module sequentially until a step that is labelled as correct cannot be verified (using a proof search depth of four), in which case we move on to the next dialog. We then compared the results of the automated proof step analysis with the original correctness judgements given by the tutors.

Table 11.1 shows the result of running the algorithm on the corpus. Of the 116 correct steps, 113 (97.4%) were correctly verified and we correctly classify 141 out of the 144 steps (97.9%) as correct or wrong. For the remaining three steps the verification fails. This is due to our restriction of the search space to either forward or backward search (at the assertion level) for efficiency reasons. These steps are not captured by the current filter functions because they require a mixture of both. It would be straightforward to allow such a mixture, and we could easily implement a new filter function which would capture these steps. However, we feel that these steps are exceptions and decided for the more efficient variant. All steps in the example dialog (c.f. Figure 11.10) are correctly classified as valid by our verification module (used with proof depth four) in less than 0.01 time on a standard PC.

The main reason for the efficiency and the simplicity of our algorithm lies in the fact that we directly search at the assertion level. Consequently, a small search depth is

correctly rejected:	28/28
correctly accepted:	113/116
wrongly accepted:	0/144
not verified:	3/144

Table 11.1: Evaluation of the algorithm

sufficient to verify a correct proof step. Indeed, a search depth of four already suffices to obtain the results shown above. In the example illustrated in Figure 11.10, it is even the case that a depth limit of 2 would suffice to find the reconstruction. The nine student steps (dark boxes) correspond to 14 reconstruction steps, which shows the close correspondence between the reconstruction and the original input. Here, the formulas given by the student are shaded. The number of assertion level steps required (13, excluding the automatic *Close* steps) is still comparable to the number of proof steps as uttered by the student in the original dialog (10), which provides evidence that the ΩMEGA assertion level proof is at a suitable level of granularity. In particular this allows for a further analysis of proof given by the student.

11.6 Possible Extensions

In the sequel, we discuss two extensions which are essential for a tutor system, namely *error detection* and the generation of *hints*. We show how these extensions can easily be integrated within our framework.

11.6.1 Error Detection

During a tutorial session, not all student steps are correct. A first kind of error, which is easy to detect and to analyze, is a syntax error. Such errors can already be resolved by error correcting parsers [SA99]. Similarly, type errors, such as using conjunction instead of intersection on sets, are easy to find. However, there are also semantic errors, which are much harder to detect and analyze. Suppose that in the verification phase no MPS consistent with the filter corresponding to the input proof step can be derived within the given bound. There are two possible causes: (i) The step was logically correct, but needs more than n deduction steps to be verified, (ii) the step was wrong. So far, the verification component can only provide the information that it was unable to verify the proof step.

In the sequel, we discuss three approaches which can simply be integrated into our framework and which provide further information.

The Buggy Rule Approach

A frequent approach to detect student errors consists of analyzing common mistakes and formulate so-called *buggy rules* [BV80]. Such an approach is for example taken in LispIts [FAR84], Buggy [BB78], Slopert [Zin06], AlgeBrain [SRA99], and ActiveMath [Mel05]. The general idea is to extend the inferences $\mathcal{I}$ of the current theory by inferences $\mathcal{I}_f$ representing typical errors in the domain. The approach fits in our framework (without any further work) as follows:

When the filter Θ_c does not find any consistent successor states in step (ii) of verification, the prover attempts to verify the proof step using the extended set of rules $\mathcal{I} \cup \mathcal{I}_f$.

If this is successful then the student has made a mistake, and the prover can report the concrete error.

For example, in the case of utterance **S10** in Figure 11.2(b), the prover will not find any consistent successor states for the call $\mathbf{hence}(s,r) \in R \circ S$ starting from the proof state generated by $\mathbf{let}(s,r) \in (R \circ S)^{-1}$. However by adding either the inference $(x,y) = (y,x)$ or $(x,y) \in R \Leftrightarrow (x,y) \in R^{-1}$ to the set of rules, consistent successor nodes can be found. The tutorial environment can then check in the proof object which inference from $\mathcal{I}_f$ has been used, and offer suitable feedback.

Although it is sometimes possible to collect common student mistakes by buggy rules, the approach has two disadvantages: Buggy rules are usually labor-intensive to specify, and they cannot deal with unforeseen misconceptions. Moreover, they are also cases in which the student's input representation is not correct. In such cases, the errors need to be detected within the parser grammar or by type inference, that is, before the actual reconstruction starts.

Integration of Model Generators

A variety of tools has been developed for finding finite models of first order logic (FOL) formulas, such as Paradox [CS03], Finder [Sla94], Sem [ZZ95], Mace4 [McC03], Falcon [Zha96], Satchmo [ABEG95], and Fmset [BH98], to name a few. Given for example the theory of groups and the assertion that all groups are commutative, they are able to produce a counter-model of the assertion, i.e., a non-commutative group. This is done by providing an interpretation for the involved function symbols which makes the assertion false, which can be understood as a group table. Similarly, counter-models can be generated for other theories, such as the theory of binary relations. Note that in contrast to the verification of a proof step, the counter-model provides us the information that the given step is wrong. The approach requires a simple syntax translation of the current proof state and the theory in the format of the model finder.

Similarity Measures and Weak Filters

Another possibility is to keep relying on the forward search and comparing the utterance to the generated proof states, thereby trying to identify a proof state containing an assertion that comes very close to the utterance. This can naively be done by weakening the filter functions, for example by allowing for non-admissible substitutions[3]. More generally, we can compute the distance between two formulas in a systematic way by computing the tree distance between the formulas. Already in 1979, the nowadays generally accepted similarity measure for trees, the so called *tree-edit distance metric* (ted), has been introduced by Tai [Tai79] and a relative efficient algorithm proposed by Klein [Kle98]. An overview of current literature about tree edit distance functions and several variants can be found in [Bil05]. For the case of algebraic manipulations, some first attempts towards the realization of similar ideas (in a much less systematic way) have been made in [Bou07] and [PJ06].

11.6.2 Generating Hints

In a number of situations, for example after repeated student errors, the tutor can decide to offer a hint to the student a long period of silence, or upon a direct request. When

[3]This is already sufficient to detect errors like those in utterance **S10**.

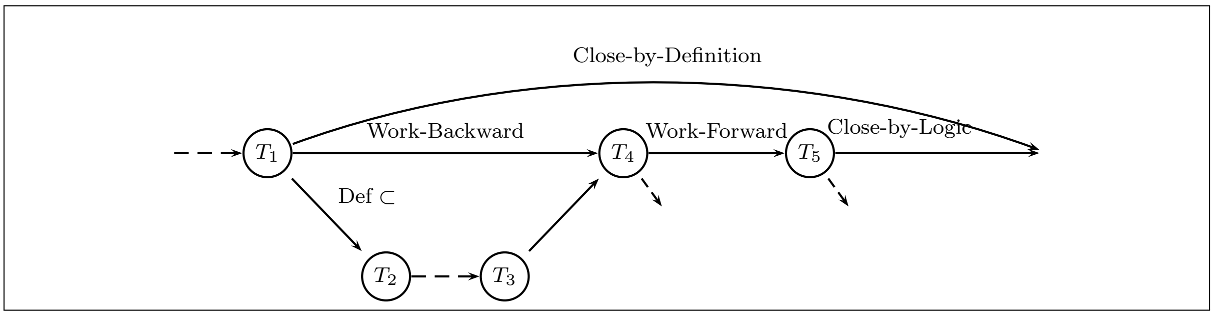


Figure 11.11: Hierarchical proof plan completing the proof of task T_1

applying a socratic strategy, a tutor gives hints through which students achieve self-explanation rather than simply providing answers. Students are encouraged to think for themselves and construct their own solution to the task at hand rather than relying on a solution presented to them by the tutor.

In [HMAE96] it has been shown that the content of hints is context sensitive, i.e., depends on all interactions performed so far. A natural extension of our dynamic analysis approach is to also dynamically generate hints once the student gets stuck. This can be done by completing the proof the student has started and subsequently generating suitable hints from the resulting solution. Hints generated in this fashion respect the steps previously entered by the student and thus contribute to the socratic teaching method.

In keeping with the socratic approach to tutoring, the student should receive hints which are initially as abstract as possible. This leaves the student to perform the actual concrete steps that the hint has requested, leading to better knowledge construction. If the student is still stuck, subsequent hints should refer to smaller subtasks of the proof, becoming increasingly close to the fully-specified assertion level step. In the sequel, we show how this idea can naturally be realized by relying on the hierarchical structure of the completed proof in the form of the $\mathcal{PDS}$. The following example illustrates the mechanism:

Example 11.6.1. *Within the running example, suppose that the student starts the proof applying the definition of set equality, yielding two subtasks*

$$T_1 :(R \circ S)^{-1} \subset S^{-1} \circ R^{-1} \tag{11.1}$$
$$T_1' :S^{-1} \circ R^{-1} \subset (R \circ S)^{-1} \tag{11.2}$$

and requests a hint for the task T_1. A possible completion of the proof, encoded in the strategy "Close-by-Definition", consists of expanding all definitions and then using logical reasoning to complete the proof. "Close-by-Definition" relies on three sub-strategies, "Work-Forward", "Work-Backward" and "Close-by-Logic". "Work-Forward" works on the assumptions of the current task and mainly expands definitions. "Work-Backward" tries to simplify the current goal by applying definitions. "Close-by-Logic" applies logical reasoning, such as performing case splits, to close the task it was applied to.

The resulting hierarchical proof object represented in the $\mathcal{PDS}$ is shown schematically in Figure 11.11. The task T_1 has three outgoing edges, the topmost two corresponding to a strategy application and the lowermost corresponding to an inference application, respectively. Internally the edges are ordered with respect to their granularity. In the example the most abstract outgoing edge of T_1 is the edge labeled with "Close-by-Definition", followed by the edge labeled with "Work-Backward", both representing strategy applications. The edge with the most fine-grained granularity is the edge labeled with "Def ⊂" and represents an inference application.

A $\mathcal{PDS}$-view extracts from a given $\mathcal{PDS}$ a proof at a specific level of granularity by selecting one proof step, i.e., one outgoing edge, for each node. Thus a $\mathcal{PDS}$-view is a part of the $\mathcal{PDS}$ without hierarchies.

By selecting the edges "Work-Backward", "Work-Forward", and "Close-by-Logic", we obtain a flat graph connecting the nodes T_1, T_4, and T_5. A more detailed proof-view can be obtained by selecting the edge "Def $\subset$" instead of "Work-Backward". In this case the previously single step leading from T_1 to T_4 is replaced by the subgraph via T_2 and T_3.

After completing the proof attempt, the resulting $\mathcal{PDS}$ can be used to generate several hints for the proof situation the student is in. For the hint generation there are two important dimensions to be considered: the vertical and the horizontal dimension. Once the vertical dimension has been fixed by selecting an appropriate $\mathcal{PDS}$-view, the resulting flat proof of a specific granularity can be analyzed to extract a sequence of hints, for example by analyzing any intermediate state between the student state and the goal, such as the first step. The hint obtained in this way can then be verbalized and be shown to the user. This decision is based on the current user model and particularly on its mastery level for the corresponding concept: the higher this value the lower the level of detail used to display the hint.

In the situation above, we assume to select exemplary the lowest level of granularity, and the first proof state to extract a hint. This already allows the generation of three hints, such as

- "Try to apply Def $\subset$"
- "Try to apply Def $\subset$ on $(R \cup S) \circ T \subset (T^{-1} \circ S^{-1})^{-1} \cup (T^{-1} \circ R^{-1})^{-1}$"
- "By the application of Def $\subset$ we obtain the new goal $(x, y) \in (R \cup S) \circ T \Rightarrow (x, y) \in (T^{-1} \circ S^{-1})^{-1} \cup (T^{-1} \circ R^{-1})^{-1}$"

Selecting a more abstract level would result in hints like "Try to work backward from the goal", or "Try to apply definitions on the goal and assumptions".

11.7 Discussion

It turned out to be very useful to search for the proof directly at the assertion level, because it reduces the gap (and thus the search effort) between the informal proof steps given by a student and the formal proof object reconstructed by the domain reasoner. This is in contrast to the development of classical search based theorem provers and the corresponding investigations of logical calculi, which are mainly driven by correctness, completeness and efficiency issues. They do not operate on a "human-oriented level", but almost always on the "machine code" of some particular logical calculus, such as resolution. Hence they are much more difficult to use to determine the general appropriateness of a proof step proposed by a learner in a tutorial context, even though this is still possible (see [Gol73] for an approach).For instance, teachers of math may reject a proof step even though it is logically correct as it may lack other desirable properties, in particular, if the steps are of inappropriate size or just irrelevant. While there are algorithms to transform low level proofs, such as resolution, to natural deduction or even the assertion level, they cannot deal with incomplete information, such as underspecified subgoals or assumptions. Note that by leaving out a subgoal, the resulting statement becomes logically wrong and is thus not checkable. The possibility to be able to deal with such problems is thus a key novelty of our work. Due to its abstract level, the proof object provides an understanding of the student step which in turn is useful for further analysis of the step, such as granularity or relevance. Another motivation for high-level reasoning is efficiency: Resolution of ambiguity and underspecification seems easier to be conducted at the abstract rather than at the detailed level. This is because each choice at the abstract level usually corresponds to several choices at a lower level. The use of uninformed search techniques at the low level to analyze and reconstruct abstract steps can thus quickly lead to unacceptable system response times for complex problems.

Also the compilation technique proved to be very useful. While in a previous version the verification needed on average 1s for each proved student step with a search time of several seconds for wrong steps, the speedup by a factor of *more than thousand* makes the verification a push-button technology.

Finally, let us discuss the importance of the strategy language within our scenario. The fact that the complex verification strategies could easily be specified within the language in a compact way illustrates the power and flexible application areas of the developed language. Beyond that, the developed strategy language provides an important means to express domain knowledge, such as solution strategies, in a clean compact way, as well as with a precise semantics. As it is mostly independent of the underlying programming language, it can also be used by non experts.

11.8 Related Work

A classification and a detailed overview of relevant tutor systems has already been given in Section 2.4.2. Within our context, we are mainly interested in systems that support tutoring in logic or more abstract mathematics, and that are built on a solid logical foundation. We subsequently compare our work with representatives of these categories.

APROS: The APROS project (see [SRL$^+$06] for an overview) provides an integrated environment for strategic proof search and tutoring in natural deduction calculi for predicate logic. It consists of four modules: the *proof generator* which implements strategic proof search based on the intercalation calculus, the *proof lab*, which builds the interface to the students, the *proof tutor*, which generates hints for students that are stuck, and a web-based course containing additional learning material. Proofs are represented in a Fitch-style diagram and constructed by adding/removing steps to the diagram. This means that the student enters stepwise the proof at the calculus level, and that the performed steps can therefore immediately be checked. If a student requests a hint, the proof generator initiates the construction of a complete proof, which the tutor analyzes to extract a hint. The first hint provided at any point in the proof is a general strategic one, and subsequent hints provide more concrete advice as to how to proceed. The last hint in the sequence recommends that the student take a particular step in the proof construction.

Compared with APROS, which focuses on the teaching of one particular logical calculus (without equality), the main difference with our approach is that we focus on the teaching of more abstract assertion level proofs, which are rather independent of a particular logical calculus. Proofs are essentially constructed in a declarative proof language in the form of proof sketches. If the information provided by the student is complete and correct, the verification is just a simple checking, as in the case of APROS. However, within our setting, this is not the typical situation. Rather, it is common that the information provided by the student is incomplete, as humans typically omit information they consider unimportant or trivial. Therefore, reconstruction of the missing information is necessary, as well as an analysis of the complexity of this information.

The generation of hints is similar to our approach in the sense that (i) it is dynamic and based on a completion of the student's proof attempt, and (ii) that it can be provided at several levels of granularity. However, we do not focus on a particular calculus, neither on a fixed proof strategy. Therefore, our framework requires sophisticated theory management, including the specification of domain-specific problem solving knowledge in the form of assertions and proof strategies. Within our framework, this has been realized using sophisticated knowledge representation techniques, which were developed in the context of proof planning within the last decades. In addition, it requires a student model to maintain the knowledge the student is supposed to know and to learn, as developed by Schiller in [Sch10].

EPGY: The EPGY theorem proving environment aims to support "*standard mathematical practice*" both in how the final proofs look as well as the techniques students use to produce them (see [SN04] p. 227). To verify the proof steps entered by a student, EPGY relies on the CAS MAPLE and the ATP OTTER. The system is domain independent in the sense that the course authors can specify the theory in which a particular proof exercise takes place. Proof construction works by selecting predefined rules and strategies from a menu, such as definition expansion or proof by contradiction, or by entering formulas. For computational transformations, a so-called derivation system is provided. Once a statement is entered, the student selects a set of justifications that he thinks is sufficient to verify the new statement. The assumptions together with the goal and implicit hidden assumptions are then sent to OTTER with a time limit of four to five seconds to verify the proof step.

Compared to our approach, the main similarities are that the system aims at teaching ordinary mathematical practice independent of a particular calculus. Moreover, it uses a theorem prover as domain reasoner to dynamically verify statements entered by a student. The authors acknowledge that the use of a classical ATP to verify proof steps has the following drawbacks ([SN04] p. 253-254): (i) "One weakness of the Theorem Proving Environment is that, like most computer-based learning tools, it does not easily assess the elegance and efficiency of the student's work". (ii) "In the current version of the Theorem Proving Environment, students are not given any information as to why an inference has been rejected. Students are told generally that an inference may be rejected because it represents too big a step of logic, because the justifications are insufficient to imply the goal, or because the goal is simply unverifiable in the current setting. From our standpoint, Otter's output is typically not enough to decide which is the reason of failure".

In contrast, our approach relies on using the assertion level as a basis to verify statements uttered by a student. This results in an abstract proof object, which can further be analyzed, for example with respect to granularity, as done by Schiller [Sch10], or to extract hints on how to proceed if the student gets stuck. In particular, the problem whether specified assertions were used in the derivation can trivially be solved. We believe that limiting the runtime of OTTER does not reveal any information about the complexity of a particular proof step. While it would also be possible to analyze the resulting proof object, we believe that it is not at an appropriate level of granularity and does not reflect a human-style of proof construction. Even for natural deduction calculi, a recent investigation [SBdV06] into the correspondence between human proofs and their counterparts in natural deduction points out a mismatch with respect to their granularity.

Moreover, our approach is more flexible with respect to the following aspects: (i) Due to the use of a proof language, the student is more flexible in entering the solution. (ii) It is compatible with the buggy rule approach. Note that this is not the case for classical automated reasoners, in which an inconsistent theory makes everything provable. In contrast, our approach allows for a full control over buggy rules, such as limiting their application to a single step. (iii) Our approach supports incomplete information such as a missing subgoal. Note that by leaving out such a subgoal, the resulting proof obligation becomes unverifiable and can therefore not be supported by a classical ATP. Finally, our approach is extensible and supports the specification of domain-dependent proof strategies, as well as checking whether a particular step can be checked by a specified proof strategy. This is not possible in the work cited above.

TUTCH: TUTCH (see [ACP01]) is a proof checker that was originally designed for natural deduction proofs in propositional logic. However, it was later extended to also feature constructive first order logic to support human oriented proof steps. To that end, a simple proof language that allows steps at the assertion level was developed, as well as proof strategies that allow for an efficient proof checking for proofs within that language. This is similar to our approach, which also relies on a proof language as well as a dynamic reconstruction of the proof steps. Because of these similarities, we focus on the details of the proof language and the strategies to verify the proof steps. The proof language features the following proof commands:

- The `triv` construct justifies a proposition that follows "trivially" by the application of some logical rules or a local lemma. It is implemented by *finishing*, which applies right rules as well as the rules $\bot_L$ and $\wedge_L$. The right rules have been modified to not modify the available assumptions: $\Rightarrow_R$ does not introduce a new assumption A when applied to a goal $A \Rightarrow B$, and parameters introduced by $\forall_R$ are not allowed to be used as witnesses for an existential variable of an assumption. Within our setting, this strategy corresponds to a repeated application of the axiom rule with the restriction that the premise must be instantiated on the left-hand side of the sequent (possibly deep), and conclusion must be instantiated on the right-hand side of the sequent (possibly deep), and disallowing parameters of the goal in substitutions.

- The `lemma` command allows the application of a lemma that is either globally available or locally inside the current proof context to derive a new fact l. The specified lemma or a formula of the current context is chosen. The fact l can be decomposed using $\Rightarrow_R$ and $\forall_R$, or focusing can be applied to the selected lemma. Focusing decomposes a formula by applying left rules, while preconditions arising due to $\Rightarrow_L$ are solved using the finishing strategy. The focusing strategy is designed to capture the idea that in mathematical practice, one instantiates all universal quantifiers and all preconditions at once when applying a lemma.

  Compared to our approach, the lemma command corresponds to a forward step as modeled by **hence**. However, our approach is more general as it allows several lemmas to be applied at once to derive a new fact. Focusing corresponds to the derivation of an inference rule, with the difference that the derivation can be stopped at any time. In contrast, we decide in favor for a specific decomposition, which is computed once and for all and then cached. This has the advantage that the resulting proof becomes more abstract, and that it becomes possible to attach control information to the resulting macro step.

- The `assume` command introduces a new assumption to the proof context. It is verified as follows:

  (i) If the goal is of the form $A \Rightarrow C$ and the assumption to be checked is A, it adds the hypothesis A and introduces C as the new goal.

  (ii) If the goal is of the form $\forall x.A$ and the assumption is x, then it introduces x as new parameter to the context.

  For introducing assumptions, our approach is more flexible as it can also be used if the goal is not an implication or universally quantified. As an example of this situation consider the first step of Figure 11.2(a).

- The `case` syntax formalizes case distinction and existential elimination. For case distinctions, the verification is divided into two phases: determining the split tree of a formula the case distinction is over, and checking that the stated cases cover the leaves of split tree. The split tree is the derivation obtained by repeatedly applying $\exists_L, \vee_L$ to the split formula A. Splitting $\perp$ succeeds immediately.

  In TUTCH, all cases of the case distinction need to be specified to be able to process the script. Missing a case is similar to missing a subgoal, both features are supported by our approach. However, they give rise to proof alternatives which cannot be maintained by TUTCH.

11.9 Summary

In this chapter we demonstrated that the abstract nature of the assertion level provides an efficient possibility to reconstruct ambiguous and incomplete student proof steps by a simple depth bounded forward search that cannot be automated by pure logical means (Evaluation E1, Section 1.1). A novelty of our work is that underspecified subgoals or assumptions are supported in the student's input. The underspecification is resolved if possible and propagated to the next student move otherwise. Known approaches to detect typical student errors can easily be integrated within our framework. Moreover, our abstract proof reconstructions naturally provide the possibility to further analyze a student step, e.g., to check its relevance or granularity, which is very difficult in other calculi.

12 A Theorem Prover Operating at the Assertion Level

To investigate the usefulness of proof search directly at the assertion level we take the domain of naive set theory, because : (i) It has been noted in [McM91] that “set theory is a notoriously difficult domain for automated reasoning programs”. In particular, classical reasoners based on resolution typically suffer from the lack of goal directedness and reach their limits due to the combinatorial explosion of the search space. Within set theory, this explosion occurs rather quickly as the concepts such as set equality, the subset relation, union and intersection are defined with respect to a single predicate membership. Even worse, translating such axioms into clausal normal form can create quite many redundancies in the search space, as well as obscure the logical structure of the formulas to some extend, such that subsequent proof search becomes very difficult, if not impossible (see for example [GS05]). As a consequence, even simple theorems can often not be proven automatically. Therefore, there is a high potential for assertion level proof search within this domain. (ii) In [HF96] Huang motivates proof presentation at the assertion level with an example taken from set theory. Indeed, most problems in set theory can easily be presented and naturally understood by students. (iii) There are many problems of set theory available in the TPTP problem library [Sut09]. In contrast to purely logical domains, the problems consist of a problem formulation and a set of axioms or definitions. Moreover, within the domain there are several rather simple problems as well as problems which cannot be solved by the majority of state of the art theorem provers.

12.1 The Problem Domain

The test problems we consider are universally quantified statements involving equality $=$, inequality ($\neq$), or the subset relation ($\subset$) between sets constructed with the set operations union ($\cup$), intersection ($\cap$), set-difference ($\setminus$), power set ($\mathcal{P}$), sum ($\bigcup$), and product ($\bigcap$), as well as the concept of the empty set $\emptyset$. All problems as well as the formalization of the underlying concepts are taken from the TPTP library, which is a library of test problems for automated theorem proving (ATP) systems. As a test bed we chose *all* problems within the category which are in nonclausal form and which use the following axiomatization (axiom set SET006+0.ax):

$$\forall A, B. A \subset B \Leftrightarrow \forall x. x \in A \Rightarrow x \in B \tag{12.1}$$
$$\forall A, B. A = B \Leftrightarrow A \subset B \wedge B \subset A \tag{12.2}$$
$$\forall X, A. X \in \mathcal{P}(A) \Leftrightarrow X \subset A \tag{12.3}$$
$$\forall x, A, B. x \in A \cap B \Leftrightarrow x \in A \wedge x \in B \tag{12.4}$$
$$\forall x, A, B. x \in A \cup B \Leftrightarrow x \in A \vee x \in B \tag{12.5}$$
$$\forall x. x \notin \emptyset \tag{12.6}$$
$$\forall b, A, E. b \in E \backslash A \Leftrightarrow b \in E \wedge b \notin A \tag{12.7}$$
$$\forall X, A. X \in \bigcup(A) \Leftrightarrow \exists Y. Y \in A \wedge X \in Y \tag{12.8}$$
$$\forall X, A. X \in \bigcap(A) \Leftrightarrow \forall Y. Y \in A \Rightarrow X \in Y \tag{12.9}$$

Moreover, we exclude problems that require proper equality handling (= is formalized as predicate set_equal). This way, we obtain 44 test problems, shown in Table 12.1. Each entry of the table contains the name of the problem, a difficulty rating (taken from the TPTP version 4.0.1), and the problem statement. The difficulty rating is a real number in the range $[0, 1]$, where 0.0 means that all state-of-the-art ATP systems can solve the problem (the problem is easy), and 1.0 means no state-of-the-art ATP system can solve the problem (the problem is hard).

Problem	Difficulty	Formalisation
SET002+4	0.48	$\forall A. A \cup A = A$
SET012+4	0.65	$\forall A. \forall E. A \subset E \Rightarrow E \backslash (E \backslash A) = A$
SET013+4	0.52	$\forall A. \forall B. A \cap B = B \cap A$
SET014+4	0.48	$\forall A. \forall X. \forall Y. X \subset A \wedge Y \subset A \Leftrightarrow X \cup Y \subset A$
SET015+4	0.52	$\forall A. \forall B. A \cup B = B \cup A$
SET019+4	0.00	$\forall A. \forall B. A \subset B \wedge B \subset A \Rightarrow A = B$
SET027+4	0.04	$\forall A. \forall B. \forall C. A \subset B \wedge B \subset C \Rightarrow A \subset C$
SET062+4	0.04	$\forall A. \emptyset \subset A$
SET063+4	0.22	$\forall A. A \cap \emptyset = \emptyset$
SET162+4	0.43	$\forall A. A \cup \emptyset = A$
SET199+4	0.52	$\forall A. \forall X. \forall Y. A \subset X \wedge A \subset Y \Leftrightarrow A \subset X \cap Y$
SET143+4	0.83	$\forall A. \forall B. \forall C. A \cap B \cap C = A \cap B \cap C$
SET148+4	0.43	$\forall A. A \cap A = A$
SET155+4	0.87	$\forall A. \forall B. \forall E. A \subset E \wedge B \subset E \Rightarrow E \backslash (A \cup B) = (E \backslash A) \cap (E \backslash B)$
SET156+4	0.87	$\forall A. \forall B. \forall E. A \subset E \wedge B \subset E \Rightarrow E \backslash (A \cap B) = (E \backslash A) \cup (E \backslash B)$
SET159+4	0.78	$\forall A. \forall B. \forall C. A \cup B \cup C = A \cup B \cup C$
SET169+4	0.87	$\forall A. \forall B. \forall C. A \cap (B \cup C) = A \cap B \cup A \cap C$
SET171+4	0.83	$\forall A. \forall B. \forall C. A \cup B \cap C = (A \cup B) \cap (A \cup C)$
SET347+4	0.17	$\bigcup(\emptyset) = \emptyset$
SET355+4	0.17	$\forall A. \forall X. X \in A \Rightarrow X \subset \bigcup(A)$
SET358+4	0.96	$\forall A. \forall B. \bigcup(A) \cup \bigcup(B) = \bigcup(A \cup B)$
SET366+4	0.09	$\forall A. \emptyset \in \mathcal{P}(A)$
SET372+4	0.96	$\forall A. \forall B. \mathcal{P}(A \cap B) = \mathcal{P}(A) \cap \mathcal{P}(B)$

SET595+4	0.78	$\forall A.\forall E.A \subset E \Rightarrow (E\backslash A) \cup A = E$
SET602+4	0.35	$\forall E.E\backslash E = \emptyset$
SET603+4	0.48	$\forall E.E\backslash\emptyset = E$
SET687+4	0.00	$\forall A.A \subset A$
SET689+4	0.09	$\forall A.\forall B.\forall C.A \subset B \wedge B \subset C \wedge C \subset A \Rightarrow A = C$
SET690+4	0.91	$\forall A.\forall B.\forall C.A\cap B\cup C = A\cap(B\cup C) \Leftrightarrow C \subset A$
SET691+4	0.13	$\forall A.A \subset \emptyset \Leftrightarrow A = \emptyset$
SET692+4	0.57	$\forall A.\forall B.A = A \cap B \Leftrightarrow A \subset B$
SET693+4	0.57	$\forall A.\forall B.A = A \cup B \Leftrightarrow B \subset A$
SET694+4	0.83	$\forall A.\forall B.\mathcal{P}(A) \cup \mathcal{P}(B) \subset \mathcal{P}(A \cup B)$
SET695+4	0.52	$\forall A.\forall B.\forall E.A \subset E \wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow E\backslash B \subset E\backslash A)$
SET696+4	0.48	$\forall A.\forall E.A \subset E \Rightarrow (E\backslash A) \cap A = \emptyset$
SET697+4	0.65	$\forall A.\forall B.\forall E.A \subset E \wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow A \cap (E\backslash B) = \emptyset)$
SET698+4	0.87	$\forall A.\forall B.\forall E.A \subset E \wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow (E\backslash A) \cup B = E)$
SET699+4	0.57	$\forall A.\forall B.\forall E.A \subset E \wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow A \cap (E\backslash B) \subset E\backslash A)$
SET700+4	0.52	$\forall A.\forall B.\forall E.A \subset E \wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow A \cap (E\backslash B) \subset B)$
SET701+4	0.74	$\forall A.\forall B.\forall C.\forall E.A \subset E\wedge B \subset E \Rightarrow (A \subset B \Leftrightarrow A \cap (E\backslash B) \subset C \cap (E\backslash C))$
SET702+4	0.61	$\forall A.\forall B.\bigcap(A) \cap \bigcap(B) \subset \bigcap(A \cap B)$
SET704+4	0.26	$\forall A.\forall X.X \in A \Rightarrow \bigcap(A) \subset X$
SET705+4	0.04	$\forall A.A \in \mathcal{P}(A)$
SET706+4	0.91	$\forall A.\forall B.\forall C.C \subset B \wedge B \subset A \Rightarrow A\backslash C = (B\backslash C) \cup (A\backslash B)$

Table 12.1: Test problems from TPTP for the evaluation of the prover

12.2 The Setting

The search for a proof is directly performed on top of the assertion interface, with the following parameters:

- The axiom rule (see 9.16) is restricted to literals. Application of the axiom rule automatically generates alternatives for all possible closures as well as an alternative where the literal is not used for closure.

- Literals that have been generated are maintained in a separate list to speed up the proof search slightly by avoiding to check the applicability of expansion rules. Thus, a proof state is a list whose elements have the following form:

$$l_1, \ldots, l_n \triangleright \Gamma \vdash \Delta, l_{n+1}, l_m \tag{12.10}$$

- All initial axioms of the theory are lifted to the level of inference rules.

- To avoid shifting of formulas that contain negations from Γ to Δ we add rules to expand negated occurrences of definitions. Note that these rules have exactly the same effect as shifting the formula to the other side and applying the corresponding rule.
- Free variables are introduced for unknowns, we use the δ^{+^+} (see [BHS93]): when eliminating the universal quantification for some succedent formula $\forall x.F$, the rule allows to take the same Skolem function for all formulas that are equal modulo α-renaming to $\forall x.F$. Secondly, the arguments to the Skolem function are only all free variables that actually occur in $\forall x.F$.
- Whenever possible, the goal formula is expanded. Otherwise, we expand the leftmost literal in the antecedent, if this is not possible, we test the axiom rule and shift it to the literal set otherwise.
- We use only rules that introduce terms that are smaller with respect to the symbol ordering induced by the definitions, i.e., the defined concept needs to be instantiated.
- If an agenda cannot be closed, it is discarded and the next alternative is explored.
- For backward application, only the conclusion of the rule is instantiated.

For example, for the definition of subset, we use the following rules:

$$\subset_1 \frac{\begin{array}{c}[x \in A] \\ \vdots \\ x \in B\end{array}}{A \subset B\{*\}} \text{NEW(X)} \qquad \subset_2 \frac{A \subset B\{*\} \quad x \in A}{x \in B} \qquad \subset_{\neg 1} \frac{x \in A \quad \neg x \in B}{\neg A \subset B\{*\}}$$

$$\subset_{\neg 2} \frac{\neg A \subset B\{-\} \quad x \in A}{x \notin B} \text{NEW(X)}$$

Let us stress the fact that formulas that introduce meta-variables must not be deleted to guarantee that sufficiently many instances can be generated during the problem solving process. Fairness is implemented by putting the formulas at the end of the antecedent afterwards.

12.3 Benchmarks

The following table summarizes the results of running the assertion level prover as well as a reference prover on an Intel(R) Core(TM)2 Duo CPU with 2.50GHz. In the experiments we measure both the runtime as well as the number of clauses/sequents generated during the proof attempt. We use the symbol ∞ to indicate that a prover was not able to find the proof within a time limit of 300 seconds. Let us stress here that even if the number of the generated sequents is very small in the case of the assertion level prover, this number does not necessarily coincide with the proof length of the final proof. If several alternatives need to be explored, the number of the actual proof is strictly smaller than this number. As a reference point, we use the prover E [Sch02, Sch04], which is currently one of the most efficient theorem provers based on a modified version of the superposition calculus for equational clausal logic as described in [BG94].

TPTP	E 0.999-006 Longview2		ΩMEGA (fv,atom)	
Problem	Time [s]	Clauses	Time [s]	#Seq.
SET002+4	7.628	416543	0.007	11
SET012+4	∞	–	0.006	23
SET013+4	12.233	532208	0.005	17
SET014+4	∞	–	0.007	53
SET015+4	9.161	468256	0.005	17
SET019+4	0.028	14	0.006	34
SET027+4	0.024	542	0.006	17
SET062+4	0.012	58	0.006	3
SET063+4	0.016	345	0.006	10
SET162+4	7.368	392258	0.005	11
SET199+4	∞	–	0.006	50
SET143+4	∞	–	0.006	33
SET148+4	12.577	543404	0.006	9
SET155+4	∞	–	0.006	39
SET156+4	∞	–	0.007	43
SET159+4	∞	–	0.006	27
SET169+4	∞	–	0.006	42
SET171+4	∞	–	0.006	38
SET347+4	0.016	320	0.006	8
SET355+4	0.164	6560	0.006	10
SET358+4	∞	–	0.007	82
SET366+4	0.020	65	0.005	4
SET372+4	∞	–	0.006	56
SET595+4	∞	–	0.006	22
SET602+4	0.016	344	0.005	10
SET603+4	11.401	510846	0.005	9
SET687+4	0.004	66	0.006	3
SET689+4	0.008	87	0.005	65
SET690+4	∞	–	0.006	53
SET691+4	0.012	71	0.004	24
SET692+4	∞	–	0.002	30
SET693+4	∞	–	0.005	53
SET694+4	∞	–	0.006	23
SET695+4	∞	–	0.007	42
SET696+4	0.352	18504	0.005	16
SET697+4	0.556	17886	0.006	59
SET698+4	∞	–	0.007	144
SET699+4	0.604	19768	0.007	46
SET700+4	0.020	522	0.006	39
SET701+4	0.640	18776	0.006	62
SET702+4	∞	–	0.006	13
SET704+4	0.056	1650	0.006	11
SET705+4	0.004	65	0.006	4
SET706+4	∞	–	0.006	70

12.4 Discussion

We see that our prover has a striking performance on the given problem domain. This is remarkable, because the difficulty rankings indicate that several of the problems are very hard to solve. For example, the problems SET358+4 and SET372+4 are both ranked with 0.96. Moreover, it outperforms E with respect to runtime for most of the examples. While the runtime is an indicator for the strength of the prover, what is equally important within the interactive setting or the domain of tutoring is the number of generated clauses/sequents. Indeed, we observe that the search space traversed by the assertion level prover is much smaller than the search space traversed by the classical automated theorem prover. This gives us an intuitive explanation for the efficient search behavior. Let us stress here that the efficiency of the prover is only due to the organization of the search space, as no advanced data structures have been used within the implementation.

Let us explain the reason for the efficiency on a technical level:

Goal-Directedness: The (automatic) lifting of formulas to inference rules induces a goal directed proof search behavior. Without the use of inferences (and other heuristics), an assertion is processed independent of the structure of the current goal. This is the case in sequent and tableau systems, but also when basing the proof search on the connection method. In contrast, an inference is only applicable if the goal state matches at least one premise/conclusion. Moreover, instead of fixing the multiplicities beforehand, the use of inferences adapts the multiplicities for the assertion automatically on demand, depending on the shape of the current goal.

Without this goal-directedness, the decomposition of a formula can introduce unnecessary branching in the case that an assertion is expanded which is not needed subsequently. This introduces a dependency on the input ordering of the formulas. In contrast, the traversal of the proof search is completely determined by the structure of the goal, independent of the order of the axioms. Moreover, unnecessary axioms can be added without any impact on the proof search, as long as they do not match a goal formula. This is in contrast to standard tableau methods which usually expand all formulas in round-robin manner.

Avoiding Redundancy: In the domain of set theory, most of the axioms have the form $\forall A_1, \ldots, A_n.F$, that is several all-quantifiers occur in a row. Due to the handling at the assertion level, intermediate subformulas generated by the decomposition of the formula are not copied, reducing the number of formulas in the sequents and thus some kind of redundancy. Note that in contrast, a proof search in the sequent calculus would require to keep a copy of intermediate formulas to be able to adapt the multiplicity accordingly.

Inversion Principle: Within the theory of set theory, the inversion principle holds for many inference rules. We say that a rule is invertible if the premises of the rule are derivable iff the conclusion is. An example known from classical sequent calculus are the rules $\wedge_R$ and $\wedge_L$. Note that in the case of left rules we require that the principal formula $(A \wedge B)$ of the rule to be deleted. Similarly, the handling of set equality and many other rules obey the inversion principle. As a consequence, the search space can further be restricted by deleting principal formulas after their use.

Reductive Behavior: The choice that new formulas need to be smaller with respect to the ordering induced by the definitions has the effect that the problems are essentially reduced to propositional problems.

Interestingly, our choices are sufficient to solve all considered problems within our problem domain. The underlying principle is that the rules allow for a saturation up to redundancy as introduced by Bachmaier and Ganzinger in the context of resolution [BGML01]. The underlying idea is that a clause can be deleted if it is redundant with respect to the other clauses, corresponding to invertibility of a rule. Recently, this approach has been taken to the level of tableau calculi in [Gie06]. We are optimistic that this framework allows for a proof theoretic investigation of the calculus, however, but leave this investigation for further work.

12.5 Related Work

Muscadet: Muscadet is a prover that was initially designed to solve problems in set theory, thereby imitating human problem solving behavior. It is "a knowledge-based system [...] that uses methods which resemble those used by humans." [Pas01]. MUSCADET uses a restricted version of forward chaining after the conclusion has been fully decomposed. Thereby it uses not only calculus rules of natural deduction, but also rules that have either be put into the system or that have been generated by meta rules from axioms. MUSCADET performs equally well on our test problems and can solve each of them within $0.2s$. This is because the rules generated by MUSCADET are similar to those by our prover. It is implemented in PROLOG and makes use of the underlying highly optimized data structures. The main differences to our approach are that (i) it does not support meta-variables, (ii) does not create a human-readable output, (iii) is not as configurable as our prover: The search strategy is hard-coded into the programming language. As the search tree produced by PROLOG is implicit, the solution as well as the search tree that has been traversed cannot be analyzed. Moreover, the heuristics and inference mechanisms have not been investigated from a theoretical point of view.

Theorema: A THEOREMA prover for Zermelo-Fraenkel set theory is presented in [Win06]. As for most provers in the context of the THEOREMA project, the set theory prover "aims at generating automated proofs in human-like natural style" (see [Win06] p. 2). It is shown that the set theory prover "generates proofs within a few seconds even for examples where other provers fail completely" (see [Win06] p. 23), with runtimes between 2.4 and 155.4 seconds on a 1500 Mhz CPU. As MATHEMATICA is not open-source, we were not able to evaluate the examples on our machine. Examples from the case study above that were also proved by THEOREMA are: SET014 (3.2s), SET171 (4.0s), SET694 (5.5s), SET698 (22.7s). Let us remark that the axioms corresponding to inference rules of the provers have been removed manually from the problem description as they would allow for redundant derivations which can also be obtained by the inference rules.

Let us now explain the architecture of the THEOREMA system: Proof search is organized by so-called special provers, which are sequential collection of inference rules. These inference rules are implemented in the underlying programming language provided by MATHEMATICA. Given a set of special provers which are arranged in two levels, the special provers from level 1 are tried left to right and the first applicable inference rule is applied. If no such rule exists, all provers of the second level are tried, giving rise to alternatives. For set theory, four special provers have been developed: STP expands outermost symbols in the goals, STKBR expands set-theoretic notions in the assumptions, STC performs simplification by computation on finite sets, and STS applies special techniques for instantiation of existential formulae in the proof goal. Together with TerminalND, which detects terminal proof situations, they build the first level, and are arranged in the following order: TerminalND, STKBR, STC, STP, STS. The second level is constituted by BasicND and PND (for basic and general predicate reasoning), QR for rewriting with quantified equalities, equivalences or implications in the knowledge base, and CDP for the treatment of case distinctions. Let us remark that only STP, STKBR, STC, and STS are set theory specific, the others are of a general nature.

As in our approach, Theorema relies on strategic knowledge to solve the problems in set theory. It features higher-order logic and uses meta-variables to postpone the instantiation of γ-variables. Moreover, it supports OR-alternatives during proof search. The generated output is human-readable and the proof tree is explicitly modified by the prover. The main difference to our approach is that the provers need to be hard-coded within the programming language provided by Mathematica, while we offer an intermediate language to flexibly specify and modify search strategies. Theorema features the implicational replacement theorem to provide some kind of deep application of inference rules: If the knowledge base contains an assertion $A \Rightarrow B$, then B in the goal can be replaced by A.

12.6 Summary

In this chapter we used the TPTP problem library to show that generic assertion level reasoning is not only beneficial for proof presentation, but also for proof automation (Contribution A1(iv), Section 1.1). Our testbed from the domain of set theory contains difficult problems that cannot be solved by most state of the art reasoning systems. Moreover, our experiments indicate that the structured use of assertions results in a goal-directed search.

13 Statman Tautologies

What are the benefits we gain by relaxing the condition of the sequent calculus that inference rules can only be applied at the top-level of formulas? We will see that deep inference allows us to start a proof from subformulas, that is, from inside out. In contrast, the sequent calculus requires a subformula to be extracted by applying decomposition rules to the main formula, thereby spreading the context among different branches of the proof tree. As decomposition of a goal in the sequent calculus essentially corresponds to the computation of its clause normal form, this can result in a blowup in the number of branches, and often results in a proof state that is not human-readable. Applicability of the inference rules at any depth – as known from rewrite rules – enables shorter proofs that are not available at the sequent calculus. However, at the same time, it also increases the non-determinism in the proof search: with deep inference, the inference rules become applicable at many more positions than in the sequent calculus. Let us point out that it is generally easier to find a solution of a small depth in a search space with a high branching factor than a solution of a high depth in a search space with a small branching factor, which becomes apparent when comparing the numbers 2^{10} and 10^2.

In this chapter, we study the so-called Statman tautologies in detail: While in the sequent calculus (without cut), the proof size grows exponentially, the proof size with our deep inference mechanism only grows quadratically. Thus, we obtain an exponential speed-up. Even more interestingly, our annotation mechanism allows us to attach control information in such a way that no search is performed at all. To that end, we introduce the deep axiom rule

$$\frac{P : [Q]\{\texttt{nopobs}, *\}}{C : [Q]\{*\}} \tag{13.1}$$

and illustrate how this rule enables us to solve these tautologies – both in theory and practice. Note that this is in contrast to other deep inference mechanisms, where it is (i) not possible to attach such control information, and (ii) there are many choice points that have to be considered when searching for a proof of a Statman tautology.

Let us now turn to the definition of the n-th Statman tautology G_n ($n \geq 1$), which is defined as follows:

$$F_k = \bigwedge_{j=1}^{k} (c_j \vee d_j)$$

$$A_1 = c_1 \qquad\qquad B_1 = d_1$$
$$A_{i+1} = F_i \Rightarrow c_{i+1} \qquad\qquad B_{i+1} = F_i \Rightarrow d_{i+1}$$

$$G_n = ([[(A_1 \vee B_1) \wedge \ldots] \wedge (A_n \vee B_n)]) \Rightarrow (c_n \vee d_n)$$

Recall that due to Statman's theorem (see theorem (3.2.4)), the proof complexity is exponential in the sequent calculus without cut. To get a better understanding of these problems, we consider the cases $n = 2$ and $n = 3$ in more detail and show how the case $n = 3$ can be reduced to the case of $n = 2$, using the axiom rule (13.1).

13.1 The cases $n = 2$ and $n = 3$

For $n = 2$, the problem is given by the following formula:

$$\vdash \left[(c_1^- \vee^\beta d_1^-)^- \wedge^\alpha \left(((c_1^+ \vee^\alpha d_1^+)^+ \Rightarrow^\beta c_2)^- \vee^\beta ((c_1^+ \vee^\alpha d_1^+)^+ \Rightarrow^\beta d_2^-)^-\right)^-\right] \Rightarrow^\alpha (c_2 \vee d_2)^+ \quad (13.2)$$

The formula is already annotated with uniform types and polarities to make its behavioral structure apparent. Let us stress here the presence of the β-formulas, which give rise to branching when decomposing the formula. However, by starting the proof from the inside of the formula, we can avoid this branching and simplify the formula based on the axiom rule stated above. Let us point out that this restricted version of the axiom rule does not introduce proof obligations and can be understood as a form of simplification, as in each step some occurrence of a positive formula is replaced by *true*.

In the case where $n = 2$, we can instantiate the conclusion C of the axiom rule with $(c_1 \vee d_1)$, and find a position for the premise P that satisfies the conditions of the rule, i.e., is in α-relation, such that no β-formulas appear on the path to its minimal common parent node.

$$\vdash \left[\boxed{(c_1^- \vee^\beta d_1^-)^-} \wedge^\alpha \left((\boxed{(c_1^+ \vee^\alpha d_1^+)^+} \Rightarrow^\beta c_2^-)^- \vee^\beta ((c_1^+ \vee^\alpha d_1^+)^+ \Rightarrow^\beta d_2^-)^-\right)^-\right] \Rightarrow^\alpha (c_2 \vee d_2)^+ \quad (13.3)$$

Therefore, the application of the rule simplifies the task to

$$\vdash \left[\boxed{(c_1^- \vee^\beta d_1^-)^-} \wedge^\alpha (c_2^+ \vee^\alpha (\boxed{(c_1^+ \vee^\alpha d_1^+)^+} \Rightarrow^\beta d_2^-))\right]^- \Rightarrow^\alpha (c_2 \vee d_2)^+ \quad (13.4)$$

where we have marked two complementary subformulas, indicating the next application of the axiom rule, which yields

$$\vdash \left[(c_1^- \vee^\beta d_1^-)^- \wedge^\alpha \boxed{(c_2^+ \vee^\alpha d_2^+)}\right]^- \Rightarrow^\alpha \boxed{(c_2 \vee d_2)^+} \quad (13.5)$$

which is equivalent to G_1 (up to the name of the propositional constants) plus an additional assumption $(c_1 \vee d_1)$. Note that we have applied the axiom rule to a non-atomic formula.

Similarly, G_3 can be reduced to G_2 with an additional assumption by the following sequence of axiom rule applications:

$$\boxed{(c_1 \vee d_1)} \wedge [\boxed{(c_1 \vee d_1)} \Rightarrow c_2 \vee (c_1 \vee d_1) \Rightarrow d_2] \\ \wedge [[(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow c_3] \vee [(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow d_3]] \\ \Rightarrow (c_3 \vee d_3) \quad (13.6)$$

$$\boxed{(c_1 \vee d_1)} \wedge [c_2 \vee \boxed{(c_1 \vee d_1)} \Rightarrow d_2] \\ \wedge [[(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow c_3] \vee [(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow d_3]] \\ \Rightarrow (c_3 \vee d_3) \quad (13.7)$$

$$\boxed{(c_1 \vee d_1)} \wedge [c_2 \vee d_2] \\ \wedge \left[\left[\boxed{(c_1 \vee d_1)} \wedge (c_2 \vee d_2) \Rightarrow c_3\right] \vee [(c_1 \vee d_1) \wedge (c_2 \vee d_2) \Rightarrow d_3]\right] \\ \Rightarrow (c_3 \vee d_3) \quad (13.8)$$

$$\boxed{(c_1 \vee d_1)} \wedge [c_2 \vee d_2] \\ \wedge \left[[(c_2 \vee d_2) \Rightarrow c_3] \vee \left[\boxed{(c_1 \vee d_1)} \wedge (c_2 \vee d_2) \Rightarrow d_3\right]\right] \\ \Rightarrow (c_3 \vee d_3) \quad (13.9)$$

$$(c_1 \vee d_1) \wedge [c_2 \vee d_2] \wedge [[(c_2 \vee d_2) \Rightarrow c_3] \vee [(c_2 \vee d_2) \Rightarrow d_3]] \Rightarrow (c_3 \vee d_3) \quad (13.10)$$

which is equivalent to G_2 (up to the name of the propositional constants) with the additional assumption $(c_1 \vee d_1)$.

The example shows how the deep application of the axiom rule avoids the blowup of the problem in SK resulting from accessing the subformulas by applying the decomposition rules, which can be seen as some form of normalization[1].

13.2 The General Case

Let us now consider the general case G_n. We will prove that we can prove G_n in $O(n^2)$ steps. We use the proof idea that has already been sketched above, namely to reduce G_n to G_{n-1} using the deep axiom rule.

Theorem 13.2.1. *Given G_n, the size of the proof using the deep axiom rule* (13.1) *is $O(n^2)$.*

[1]Applying the SK rules to $\vdash A$ and collecting the branches leads to conjunctive normal form, whereas applying the SK rules to $A \vdash$ results in a disjunctive normal form

Proof. For the reduction of G_n to G_{n-1}, we have to consider the positive occurrences of $(c_1 \vee d_1)$ in G_{i+1}. By construction, G_n has the form

$$G_n = \left[\left(\left[(c_1^- \vee^\beta d_1^-)^- \wedge^\alpha (A_2^- \vee^\beta B_2^-)^-\right] \ldots \wedge^\alpha (A_n^- \vee^\beta B_n^-)^-\right)^- \Rightarrow^\alpha (c_n^+ \vee d_n^+)^+\right]^+ \quad (13.11)$$

For $i \geq 2$, each A_i as well as each B_i contains exactly one F_{i-1} and therefore exactly one occurrence of $(c_1 \vee d_1)$. As A_i and B_i have negative polarity, we have for $n \geq 2$

$$A_i = (F_{i-1}^+ \Rightarrow^\beta c_i^-)^- \quad (13.12)$$

and

$$B_i = (F_{i-1}^+ \Rightarrow^\beta d_i^-)^- \quad (13.13)$$

Therefore, by the construction of F_i, all subformulas of F_i have positive polarity. Thus, G_n contains $2(n-1)$ positive occurrences of $(c_1 \vee d_1)$, thereby enabling $2(n-1)$ axiom rule applications, as $(c_1 \vee d_1)$ is α-related to A_i for $i \geq 2$ via the smallest subformula which contains both $(c_1 \vee d_1)$ and A_i, which is

$$\left[\left((c_1^- \vee^\beta d_1^-)^- \wedge^\alpha (A_2^- \vee^\beta B_2^-)^-\right) \ldots \wedge \left(A_i^- \vee^\beta B_i^-\right)\right] \quad (13.14)$$

Moreover, as there is no β-formula this results in a rewrite rule $(c_1 \vee d_1)^+ \to \langle \text{true}^+ \rangle$.

Notation 13.2.2. *Let A be a formula with exactly one occurrence of the formula B^+. Then we denote with $A \backslash B$ the formula that is obtained by replacing B by true and applying the following simplification rules:*

$$A \wedge \text{true} \to A \qquad \text{true} \wedge A \to A \quad (13.15)$$
$$\neg\, \text{true} \to \text{false} \quad (13.16)$$
$$A \vee \text{true} \to \text{true} \qquad \text{true} \vee A \to \text{true} \quad (13.17)$$
$$\text{true} \Rightarrow A \to A \qquad A \Rightarrow \text{true} \to \text{true} \quad (13.18)$$

Note that this replacement together with the subsequent application of the simplification corresponds to the application of a resolution replacement rule that does not introduce any proof obligations followed by the application of Core*'s simplification rule.*

Using notation 13.2.2, G_n can be reduced to

$$\left[\left((c_1^- \vee^\beta d_1^-)^- \wedge^\alpha (A_2 \backslash (c_1 \vee d_1)^- \vee^\beta B_2 \backslash (c_1 \vee d_1)^-)^-\right) \ldots \wedge \left(A_n^- \backslash (c_1 \vee d_1) \vee^\beta B_n \backslash (c_1 \vee d_1)^-\right)\right] \Rightarrow (c_n \vee d_n) \quad (13.19)$$

Note that for $n \geq 2$

$$\begin{aligned} A_i \backslash (c_1 \vee d_1) &= F_{i-1} \backslash (c_1 \vee d_1) \Rightarrow c_i & (13.20)\\ &= F_{i-2}^{+1} \Rightarrow c_i & (13.21)\\ &= A_{i-1}^{+1} & (13.22) \end{aligned}$$

where F_{i-2}^{+1} denotes F_{i-2} where c_i has been replaced by c_{i+1} and d_i by d_{i+1}. In summary, we reduce G_n in $n-2$ steps to the formula

$$\left[\left(\left[(c_1^- \vee^\beta d_1^-)^- \wedge^\alpha (A_1^{+1^-} \vee^\beta B_1^{+1^-})^-\right] \ldots \wedge^\alpha ({A_{n-1}^{+1}}^- \vee^\beta {B_{n-1}^{+1}}^-)^-\right)^- \Rightarrow^\alpha (c_n^+ \vee d_n^+)^+\right]^+ \quad (13.23)$$

which is (modulo associativity) equivalent to

$$(c_1^- \vee^\beta d_1^-) \wedge G_{n-1}^+ \tag{13.24}$$

Consequently, we need

$$1 + \sum_{i=2}^{n} 2(i-1) = n^2 - n + 1 \tag{13.25}$$

axiom rule applications to solve G_n. □

On the other hand, the sequent calculus rules for α-decomposition and β-decomposition are admissible in the Core calculus (see [Aut03], p. 158). Therefore, propositional proofs in the sequent calculus can easily be simulated within Core's proof theory. Therefore, proofs in the Core calculus have in the worst case the same complexity as their counterparts in the sequent calculus. Let us point out that the possibility to find shorter proofs also introduces additional redundancy. The development of general techniques that avoid some of this redundancy together with its theoretical properties is an interesting area for future research but has not further been studied within this thesis. However, our experiments in the next section will indicate that we can take advantage of the shorter proofs in practice by relying on heuristics.

13.3 Practical Evaluation

Of course, the complexity result derived above is only a theoretical result, in the sense that there exists a proof of size $O(n^2)$. In practice, there is the question of how difficult it is to find this proof automatically. Therefore, we present empirical results based on two different implementations. Both implementations use traversal functions which keep track of the context and traverse the term structure to find two subformulas that unify and have different polarities, as described in Chapter 9. Note that we do not have implemented any indexing techniques, which could be used to speed up the traversal even more.

The first implementation is in Lisp and is purely functional. In particular, terms are naively implemented without any structure sharing. As a consequence, equality between two terms can only be decided by a complete term traversal. Hence, this could be even more improved with this kind of standard technique. To be able to judge the possibilities of our approach, we provide a second implementation, which is based on the programming language C. While it is in principle possible to write very efficient Lisp programs and for some benchmarks the runtimes are similar within both languages, a C implementation can be up to 10 times faster, depending on the problem[2]. Another advantage of C is that many high performance libraries exists. One of them is the Aterm library (see `http://www.program-transformation.org/Tools/ATermLibrary`) which has been developed for program transformation and which is used by our second implementation. The Aterm library provides maximal term sharing and therefore reduces the storage needed for the implementation. Most importantly, equality between two terms becomes check for pointer equality and can therefore be decided very efficiently. As the number of matching attempts is very high (as shown in 13.2) in our example, term-sharing pays off in practice. We compare our runtime with results from the prover E, which is a highly efficient theorem prover based on paramodulation. We also compare the result with the model finder Paradox, which performed surprisingly good on the problem. This is because

[2]see `http://shootout.alioth.debian.org/u64q/benchmark.php` for several benchmarks in both languages

Paradox implements sophisticated propagation techniques rather than transforming the original input formula.

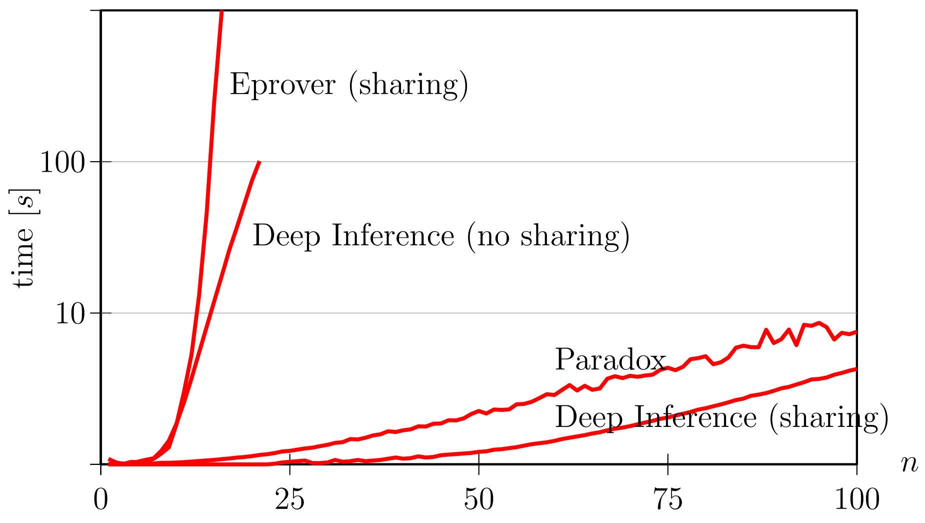


Figure 13.1: Runtime comparison for Statman tautologies

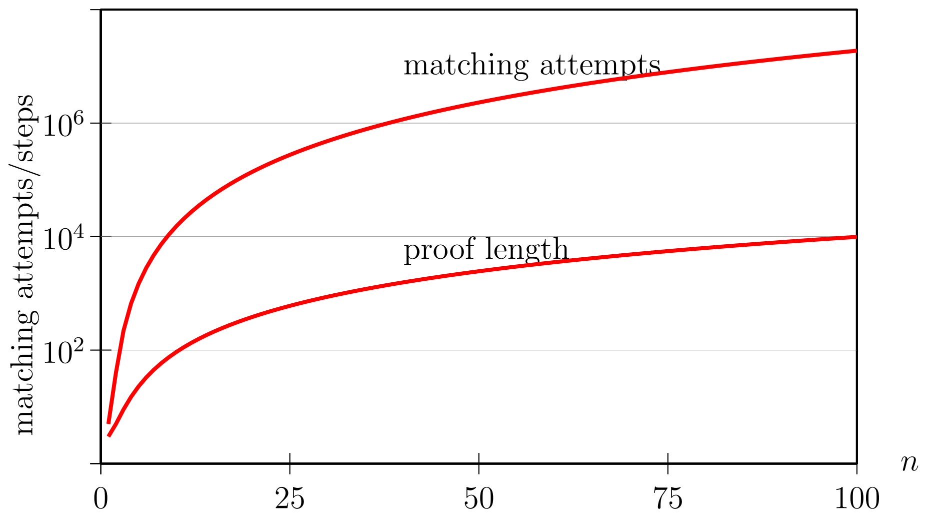


Figure 13.2: Number of proof steps and matching attempts

The exact runtimes are shown in Table 13.1.

We have also compared our runtimes with the of state of the art SAT-solver ZChaff, which is reasonable as the problem is propositional. The main problem of this comparison is that the solver needs the problem in a CNF format as input. To convert the Statman tautologies to CNF, we used the TPTP tools available from `www.tptp.org`. While the final solution phase was not measurable (< 0.001 seconds), the conversion to clause normal form was only possible for very small n on our local machine and took a considerable amount of time, as summarized in Table 13.2. For $n > 9$ the program was not able to perform the conversion due to stack overflow.

The example shows that there are situations in which it pays off to start proof search inside out. It is very surprising that our implementation outperforms the machine oriented calculi, even though our implementation has not even used all the tools of the trade. The benefits come from the fact that we do not work with clausal normal form, nor is the normal form introduced implicitly by decomposing the formulas. This allows for a simplification without the introduction of new branches or case splitting.

n	Eprover [s]	ΩMEGA (c) [s]	Paradox [s]	C Impl. + Aterm [s]
1	0.019	0.0000		0.0
4	0.030	0.011		
5	0.072	0.041		
10	0.878	0.914	0.05	
15	240.562	10.967	0.124	
20	–	74.229	0.218	0.0
50			1.27	0.21
100			6.525	3.27

Table 13.1: Runtime Comparison for the Statman tautologies

n	1	2	3	4	5	6	7	8	9	10
time [s]	0.627	0.616	0.607	0.616	0.641	0.751	1.439	5.100	–	–

Table 13.2: Time needed for conversion in clause normal form

On the other hand, the proofs that make use of deep inference are more difficult to understand. This is already the case for the Statman tautologies shown above, even though the axiom rule does not introduce any proof obligation. However, it is common for interactive theorem proving to be a combination of automated and interactive steps, and the possibility to perform powerful simplification steps will be beneficial.

13.4 Related Work - The System KSg

The proof complexity of the Statman tautologies has already been studied in the calculus of structures [GG04, BG07]. Therefore, we give a detailed description of this approach and compare it to our approach.

The calculus of structures is a proof theoretical formalism and can be seen as a generalization of the sequent calculus. The main notion is the notion of a structure, which unifies the notion of a sequent and the notion of a formula. For propositional logic, structures are defined as follows:

Definition 13.4.1 (KS)**.** Propositional variables v *and their negations* $\overline{v}$ *are* atoms. *Atoms are denoted by* $a, b, \ldots$*. The* formulas *of the language* KS *are generated by*

$$S ::= \mathrm{f} \mid \mathrm{t} \mid a \mid \underbrace{[S, \ldots, S]}_{>0} \mid \underbrace{(S, \ldots, S)}_{>0} \mid \overline{S} \tag{13.26}$$

where f *and* t *denote the units* false *and* true*,* $[S_1, \ldots, S_h]$ *is a* disjunction *and* $(S_1, \ldots, S_h)$ *a* conjunction*. Formulas are denoted by* S, P, Q, R, T, U, V *and* W*.* Formula contexts*, denoted by* $S\{\ \}$*, are formulas with one occurrence of* $\{\ \}$*, the* empty context *or* hole*, that does not appear in the scope of a negation.* $S\{R\}$ *denotes the formula obtained by filling the hole in* $S\{\ \}$ *with* R*. We drop the curly braces when they are redundant: for example,* $S[R, T]$ *is short for* $S\{[R, T]\}$*. A formula* R *is a subformula of a formula* T *if there is a context* $S\{\ \}$ *such that* $S\{R\}$ *is* T*. Formulas are syntactically equivalent modulo the smallest equivalence relation induced by the equations shown in Figure 13.3. There,* $\vec{R}, \vec{T}$ *and* $\vec{U}$ *are finite sequences of formulas,* $\vec{T}$ *is non-empty.*

The system KSg, which is proved to be sound and complete, provides the following

Associativity

$$[\vec{R},[\vec{T}],\vec{U}] = [\vec{R},\vec{T},\vec{U}] \qquad (13.27)$$
$$(\vec{R},(\vec{T}),\vec{U}) = (\vec{R},\vec{T},\vec{U}) \qquad (13.28)$$

Units

$$(f,f) = f \qquad [f,R] = R \qquad (13.29)$$
$$[t,t] = t \qquad (t,R) = R \qquad (13.30)$$

Context Closure
If $R = T$ then $S\{R\} = S\{T\}$ and $\overline{R} = \overline{T}$

Commutativity

$$[R,T] = [T,R] \qquad (13.31)$$
$$(R,T) = (T,R) \qquad (13.32)$$

Negation

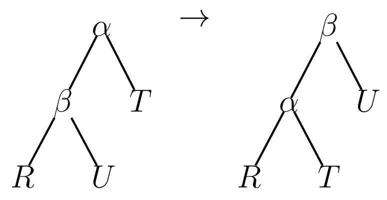
$$\overline{f} = t \qquad (13.33)$$
$$\overline{t} = f \qquad (13.34)$$
$$\overline{[R,T]} = (\overline{R},\overline{T}) \qquad (13.35)$$
$$\overline{(R,T)} = [\overline{R},\overline{T}] \qquad (13.36)$$
$$\overline{\overline{R}} = R \qquad (13.37)$$

Figure 13.3: Syntactic equivalence of formulas

inference rules for propositional logic:

$$i\downarrow\,\frac{S\{\mathrm{t}\}}{S[R,\overline{R}]} \qquad s\,\frac{S([R,T],U)}{S[(R,U),T]} \qquad w\downarrow\,\frac{S\{\mathrm{f}\}}{S\{R\}} \qquad c\downarrow\,\frac{S[R,R]}{S\{R\}}$$

The identity rule $i\downarrow$ corresponds to the axiom rule and replaces two complementary literals within a disjunction $[\cdot]$ by *true*. The switch rule s manages the disjunctive context of a conjunction in order to bring the structures closer in a disjunction, with the goal to finally apply the identity rule $i\downarrow$. Note that the rule s can be represented as



(13.38)

The weakening rule $w\downarrow$ and contraction $c\downarrow$ remove respectively duplicate a formula within a structure.

For $n = 3$, we present the proof of the Statman tautology within the system KSg.

$$\begin{array}{rc}
 & \mathrm{t} \\ \hline
i\downarrow & \boxed{[(\overline{c_1}\ \overline{d_1})\ c_1\ d_1]} \\ \hline
i\downarrow(2\times) & [(\boxed{[[c_2\ d_2]\ (\overline{c_2}\ \overline{d_2})]}\ \overline{c_1}\ \boxed{[[c_2\ d_2](\overline{c_2}\ \overline{d_2})]}\ \overline{d_1})\ c_1\ d_1] \\ \hline
s & [\boxed{(\overline{c_2}\ \overline{d_2})}\ \boxed{([[c_2\ d_2](\overline{c_2}\ \overline{d_2})]\overline{c_1}\overline{d_1})}\ \boxed{[c_2\ d_2]}]c_1\ d_1] \\ \hline
s & [(\overline{c_2}\ \overline{d_2})\ \boxed{(\overline{c_2}\ \overline{d_2})}\ (\boxed{[c_2\ d_2]\ \overline{c_1}}\ \boxed{[c_2\ d_2]\ \overline{d_1}})\ c_1\ d_1] \\ \hline
c\downarrow & [\boxed{(\ \overline{c_2}\ \overline{d_2})}\ ([c_2\ d_2]\ \overline{c_1}\ [c_2\ d_2]\ \overline{d_1})\ c_1\ d_1] \\ \hline
i\downarrow(4\times) & [([\boxed{[c_3\ d_3]\ (\overline{c_3}\ \overline{d_3})}]\ \overline{c_2}\ \boxed{[[c_3\ d_3]\ (\overline{c_3}\ \overline{d_3})]}\ \overline{d_2})\ (\boxed{[[c_3\ d_3](\overline{c_3}\ \overline{d_3})]}[c_2\ d_2]\overline{c_1}\boxed{[[c_3\ d_3](\overline{c_3}\ \overline{d_3})]}[c_2\ d_2]\overline{d_1})\ c_1\ d_1] \\ \hline
s & [\boxed{(\overline{c_3}\ \overline{d_3})}\ ([[c_3\ d_3](\overline{c_3}\ \overline{d_3})]\ [[c_3\ d_3](\overline{c_3}\ \overline{d_3})]\overline{c_2}\ \overline{d_2})(\boxed{[[c_3\ d_3]\ (\overline{c_3}\ \overline{d_3})][c_2\ d_2]\overline{c_1}[c_2\ d_2]\overline{d_1}}\ \boxed{[c_3\ d_3]})\ c_1\ d_1] \\ \hline
s & [(\overline{c_3}\ \overline{d_3})\ \boxed{(\overline{c_3}\ \overline{d_3})}([[c_3\ d_3](\overline{c_3}\ \overline{d_3})]\ [[c_3\ d_3](\overline{c_3}\ \overline{d_3})]\overline{c_2}\ \overline{d_2})(\boxed{[c_3\ d_3]}\ \boxed{[c_2\ d_2]\ \overline{c_1}\ [c_3\ d_3]\ [c_2\ d_2]\ \overline{d_1}})\ c_1\ d_1] \\ \hline
s & [(\overline{c_3}\ \overline{d_3})(3\times)\ \boxed{(\overline{c_3}\ \overline{d_3})}(\boxed{[[c_3\ d_3](\overline{c_3}\ \overline{d_3})]\ \overline{c_2}\ \overline{d_2}}\ \boxed{[c_3\ d_3]})([c_3\ d_3]\ [c_2\ d_2]\ \overline{c_1}\ [c_3\ d_3]\ [c_2\ d_2]\ \overline{d_1})\ c_1\ d_1] \\ \hline
s & [(\overline{c_3}\ \overline{d_3})\ (3\times)\ \boxed{(\overline{c_3}\ \overline{d_3})}(\boxed{[c_3\ d_3]\ \overline{c_2}}\ \boxed{[c_3\ d_3]\ \overline{d_2}})\ ([c_3\ d_3]\ [c_2\ d_2]\ \overline{c_1}\ [c_3\ d_3]\ [c_2\ d_2]\ \overline{d_1})\ c_1\ d_1] \\ \hline
c\downarrow(3\times) & [\boxed{(\overline{c_3}\ \overline{d_3})}\ ([c_3\ d_3]\ \overline{c_2}\ [c_3\ d_3]\ \overline{d_2})\ ([c_3\ d_3]\ [c_2\ d_2]\ \overline{c_1}\ [c_3\ d_3]\ [c_2\ d_2]\ \overline{d_1})\ c_1\ d_1]
\end{array} \tag{13.39}$$

Compared to our approach, the identity rule $i\downarrow$ is similar to our axiom rule, but more restricted, as it requires the two subformulas to be within a conjunction. As a consequence, it does never introduce a new proof obligation and replaces the corresponding structure by *true*. Due to this restriction, the switch rule s needs to be applied to prepare the application of $i\downarrow$.

To understand the details of the proof, we show here the first two switch steps in more detail: Writing A for $[c_2\ d_2]$ and $\overline{A}$ for $(\overline{c_2}, \overline{d_2})$ the first fifth line from above reads as

$$\overline{A} \vee \underbrace{A}_{T} \vee (\underbrace{A}_{R} \wedge \underbrace{\overline{c_1} \wedge A \wedge \overline{d_1}}_{U}) \vee c_1 \vee d_1 \tag{13.40}$$

reducing to

$$\overline{A} \vee (A \vee \overline{A}) \wedge (c_1 \wedge A \wedge \overline{d_1}) \tag{13.41}$$

which is equivalent to

$$\underbrace{\overline{A}}_{T} \vee \underbrace{(A \vee \overline{A}) \wedge \overline{c_1} \wedge d_1}_{U} \wedge \underbrace{A}_{R} \tag{13.42}$$

which reduces to

$$(A \vee \overline{A}) \wedge (A \vee \overline{A}) \wedge \overline{c_1} \wedge \overline{d_1} \tag{13.43}$$

which is equivalent to

$$(A \vee \overline{A}) \wedge c_1 \wedge (A \vee \overline{A}) \wedge \overline{d_1} \tag{13.44}$$

It has been shown that the nth Statman problem can be shown in $O(n^2)$ steps [BG07]. As before, the proof idea is to reduce the n-th Statman problem to the $n-1$ Statman problem and conclude the result by induction.

Theorem 13.4.2 ([BG07]). *Given G_n, the size of the proof in KSg is $O(n^2)$.*

More precisely, it is easy to show that the proof length in KSg is $3n^2 + 2n$. Both the switch rule and the axiom rule $i\downarrow$ are admissible in the CORE calculus. Most interestingly, our deep axiom rule is more powerful than the corresponding rule in KSg. In particular, it can directly be applied without further preprocessing steps, which are necessary in KSg. Therefore, a control of the axiom rule is much simpler. We have shown that the restricted deep axiom rule is already sufficient to solve the problem, and that the problem gets simpler after each application of this rule. This is not the case in KSg, where it is also not possible to attach control information to the rules. However, let us note here that the calculus of structures is mainly investigated from a theoretical point of view, while we are mainly interested in practical applications.

13.5 Understanding Replacement Rules

As we have seen above, the deep axiom rule, which is a special kind of resolution replacement rule, is very powerful. Therefore, it is natural to study the rule and its relation to other approaches in more detail.

13.5.1 Directions of Replacement Rules

First, we observed that in our setting above, the replacement rules were always applied in the direction from negative to positive formulas. However, in general, it is also possible to apply the rule in another direction. Therefore, it is natural to ask whether a single direction is sufficient to find all proofs. It turns out that this is not the case. The simplest example is the propositional formula

$$((P^- \vee^\beta P^-)^- \Rightarrow^\alpha P^+)^+ \tag{13.45}$$

In the example, there are four possible resolution replacement rules applicable. Restricting them to the direction to replace positive occurrences, two remain, which are of the form

$$P \to \langle \neg P \rangle \tag{13.46}$$

which transform the problem to

$$((P^- \vee^\beta P^-)^- \Rightarrow^\alpha (\neg P)^+)^+ \tag{13.47}$$

A similar situation occurs when proving Pierce's law, shown in (13.48)

$$\left[((P^- \Rightarrow^\alpha Q^+)^+ \Rightarrow^\beta P^-)^- \Rightarrow^\alpha P^+\right]^+ \tag{13.48}$$

The same result holds when restricting the replacement rules to the other direction by constructing a similar formula as the one above where positive and negative occurrences are interchanged.

To overcome this problem, we can use Schütte's decomposition rule to introduce a case split, resulting in

$$P \Rightarrow P \wedge P \Rightarrow P \tag{13.49}$$

which can be solved via the restricted replacement rules. Note that this is possible whenever a β-formula is present.

13.5.2 Deep Axiom Rule and Simplification

Another possiblity is to view the deep axiom rule – in particular in its restricted form (13.1) – as some kind of simplification rule, as the restriction guarantees that no proof obligations are introduced. In the case of propositional logic, which we will now consider in more detail, the rule is even local, in the sense that no substitution possibly affects other parts of the indexed formula tree. In the context of tableau calculi, Massacci presents the following simplification rule (see [Mas98] for details)

$$\frac{\Phi, \Psi}{\Phi[\Psi], \Psi} \tag{13.50}$$

and reports on significant speedups for some examples. The rule should be read top down and allows the expansion of a tableau node n of the form $\Phi, \Psi \cup G$ with the node $\Phi[\Psi], \Psi \cup G$, where $\Phi[\Psi]$ denotes the formula in which all occurrences of Ψ, respectively $\neg\Psi$ are replaced by true, respectively false and subsequently simplified. An example derivation which uses the simplification rule is shown below:

$$\dfrac{\dfrac{\dfrac{A, B \vee C \vdash B \vee C}{A \vdash B \vee C \Rightarrow B \vee C}\Rightarrow_R}{A \vdash ((A \Rightarrow B) \vee (A \Rightarrow C) \Rightarrow (D \Rightarrow A) \wedge (B \vee C))}\textsc{simp}}{\vdash A \Rightarrow ((A \Rightarrow B) \vee (A \Rightarrow C) \Rightarrow (D \Rightarrow A) \wedge (B \vee C))}\Rightarrow_R$$

Simplification not only leads to shorter proofs, but also fewer and smaller counter-models in the case that the conjecture is not valid, which is particularly useful in an interactive setting to be able to locate and fix the error. The following derivation gives an example:

$$\dfrac{\dfrac{\dfrac{P \vdash R \qquad \dfrac{\dfrac{P \vdash Q \qquad P, P \vdash}{P, Q \Rightarrow P}}{P \vdash \neg(Q \Rightarrow P)}}{P \vdash (R \wedge \neg(Q \Rightarrow P))} \qquad P, Q \vdash}{\dfrac{P, (R \wedge \neg(Q \Rightarrow P) \Rightarrow Q) \vdash}{(P \wedge (R \wedge \neg(Q \Rightarrow P) \Rightarrow Q)) \vdash}}}{\vdash \neg(P \wedge (R \wedge \neg(Q \Rightarrow P) \Rightarrow Q))}$$

Massacci shows that proof procedures using this rule can subsume a number of other theorem proving techniques for propositional logic, such as the unit rule of DPLL [DLL62], the β^c rules of KE [DM94], the modus ponens and modus tollens in Harp [OS88], regularity and hyper tableaux [BFN96].

Unit propagation in DPLL (which requires formulas to be in clause normal form) triggers the deletion of all clauses that contain the literal l and removes $\bar{l}$ from all remaining clauses. The β^c rules are shown below (to be read bottom up)

$$\frac{S, \beta_2, \overline{\beta_1}}{S, \beta, \overline{\beta_1}} \qquad\qquad \frac{S, \beta_1, \overline{\beta_2}}{S, \beta, \overline{\beta_2}}$$

and can also be understood as a form of simplification: Given a branching formula, e.g., $A \vee B$, and the fact $\neg A$ (respectively $\neg B$), one branch can directly be closed. Modus ponens and modus tollens in Harp are used, e.g., to simplify the two facts A and $A \Rightarrow B$ to A and B. For example, in Harp the following derivation is possible:

$$\dfrac{\dfrac{\dfrac{\dfrac{A,(A \Rightarrow B) \vee (A \Rightarrow C), D \vdash A}{A,(A \Rightarrow B) \vee (A \Rightarrow C) \vdash (D \Rightarrow A)} \qquad \dfrac{\dfrac{A,(A \Rightarrow B) \vdash B, C \qquad A,(A \Rightarrow C) \vdash B, C}{A,(A \Rightarrow B) \vee (A \Rightarrow C) \vdash B, C}}{A,(A \Rightarrow B) \vee (A \Rightarrow C) \vdash (B \vee C)}}{A,(A \Rightarrow B) \vee (A \Rightarrow C) \vdash (D \Rightarrow A) \wedge (B \vee C)}}{A \vdash ((A \Rightarrow B) \vee (A \Rightarrow C) \Rightarrow (D \Rightarrow A) \wedge (B \vee C))}}{\vdash A \Rightarrow ((A \Rightarrow B) \vee (A \Rightarrow C) \Rightarrow (D \Rightarrow A) \wedge (B \vee C))}$$

Resolution replacement rules (followed by simplification) are more general than the simplification rule of Massacci. Within our framework, the simplification rule can be modelled by the following annotated inference:

$$\frac{P\{* \vdash\}}{[P]\{* \vdash\}} \tag{13.51}$$

Due to the restriction that matchings are not overlapping, the resulting resolution replacement rule does not introduce any proof obligations and therefore behaves exactly like Massaccis simplification rule. However, in addition to that, the rule

$$\frac{[P]\{\texttt{nopob}\}}{[P]\{* \vdash\}} \tag{13.52}$$

can also simplify formulas in situations where the premise is not at top-level, but embedded within a larger formula.

To lift this rule to the first order case with free variables, one possibility is to apply the necessary substitution to the whole proof tree so that simplification becomes possible. In the case that the instantiation does not lead to a proof, the prover must backtrack and a different option is taken. Another possibility, as proposed by Giese (see [Gie01] for details), consists of decorating formulas with unification constraints. A unification constraint is a conjunction of syntactic equalities and avoids the global instantiation of the free variable. For example, $p(X) \ll X \equiv Z$ denotes that the formula $p(X)$ is only available in a way that satisfies the constraint $X \equiv Z$. Using unification constraints, the simplification rule can easily be lifted to the first order setting. A similar approach is taken by Peltier [Gal97, Pel99], which attaches constraints in the form of equalities to the formulas. This is in contrast to Giese, who separates the constraint language from the actual formula syntax. Within our framework, whenever applying a substitution, it is possible to increase the multiplicities of the involved meta-variables beforehand. Another convenient possibility is to restrict its (automatic) application to the case of so-called universal variables: These are variables for which arbitrary instances can be generated without any costs, as for example in $\forall x.P(x) \vdash P(a) \wedge P(b)$.

13.5.3 Literal Extractions

Apros implements a strategic approach of proof search in natural deduction. Among others, there is the so-called extraction strategy which is applicable if the goal occurs as a negative subformula of an assumption and is not below a negation. In this case, the idea is to apply a sequence of elimination rules to extract that occurrence, which allows the closure of the current goal. The extraction possibly yields new subgoals which subsequently need to be solved.

The extraction strategy essentially corresponds to the application of the following version of the axiom rule

$$\frac{[P]\{*\vdash, \mathit{topdown}, \mathit{stop} = \mathit{isnegation}\}}{P\{*\vdash\}}$$

The difference lies in the handling of disjunctions: If the goal G occurs as a subformula of a disjunction, for example $(A \vee B[G])^{-}$, Apros applies disjunction elimination. This results in a new subgoal G with additional assumption A. In contrast, the application of a resolution replacement rule results in the proof obligation A^{-} (giving rise to a new subgoal $\neg A$), which is β-inserted at the position of G. This can be problematic, as the following example illustrates:

$$A \vee B, A \Rightarrow B \vdash B \tag{13.53}$$

Extraction of B from the disjunction results in the new subgoal

$$A \vee B, A, A \Rightarrow B \vdash B \tag{13.54}$$

while the application of a resolution replacement rule results in the new subgoal

$$A \vee B, A \Rightarrow B \vdash \neg A \tag{13.55}$$

which is not provable (let A and B be true). Within the Core calculus, the situation above requires the explicit introduction of the case split using Schütte's decomposition rule.

Similar to Apros, the extraction strategy has been implemented in form of so-called auxiliary goal rules in the context of the SAD system [DLM99]. The main difference to the approach taken in Apros is that a negation is automatically pushed to the literals when decomposing a formula. In particular, an assumption $\neg(A \wedge B)$ introduces two case splits $\neg A$ and $\neg B$. Therefore, SAD is able to handle subformulas in assumptions below negations.

13.5.4 Matrix Calculi

Matrix or connection methods have been developed by Peter Andrews [And81] and also by Wolfgang Bibel [Bib81, Bib87]. Influenced by the work of [Wal90], they have been implemented for a variety of logics, such as intuitionistic logic, modal logics, and for fragments of linear logic (see e.g. [KO99]).

Matrix based proof search procedures work on *formula trees* and try to find a *spanning set* of *connections*, i.e., a set of α-related complementary literals such that each maximal set of α-related formulas, called *path*, contains a connection from this set. Usually, the initial multiplicities are fixed. In case no proof could be found, the multiplicities are increased and proof search is started again. Let us point our here that Core also supports the increase of multiplicities, but tries to transform the formula tree to either true^{+} or false^{-}. Moreover, the multiplicities can be increased at runtime without the need to restart the search procedure. Considering the application of a replacement rule and the insertion of a connection, we observe that both operations have similar preconditions: the corresponding positions must have opposite polarities and their labels unifiable and need to be α-related. However, while in the case of matrix calculi the α-relation is with respect to the initial formula tree, in Core the α-relation needs to hold in the free variable formula tree. In particular, due to previous replacements it is possible to apply a resolution replacement rule between literals that are β-related in the initial formula tree. Considering the initial formula tree Q, there are therefore two kinds of connections: those between nodes that are α-related in Q, and those that are β-related in Q. In the literature, these links are known as c-links and d-links and have been used to define path resolution and path dissolution (see [MR87a, MR87b]).

The main advantage of the connection method is its compactness, which allows for the development of very efficient proof search methods, while making it at the same time extremely difficult for humans to understand. The latter point concerns not only the understanding of the proofs, but also the development of implementation details. Nevertheless several systems based on the connection method have been developed, such as Setheo [LSBB92], Leancop [Ott08], and TPS [ASDP90].

13.6 Summary

In this chapter we showed that the problem class of Statman tautologies, whose solution requires $O(2^n)$ steps in the sequent calculus without cut, can be solved in $O(n^2)$ steps within our calculus due to the deep inference feature (Contribution A1(v), Section 1.1). We also showed that it is possible to find these shorter proofs in practice by constraining the application of the deep axiom rule in a specific way. Interestingly, the class of Statman tautologies can be solved without performing any backtracking, which suggests to see the restricted axiom rule as a generalized simplification rule. We also pointed out that for efficient search algorithms advanced data structures are necessary.

14 The Limit Domain

Let us now consider the so-called *limit domain* which consists of statements about the limit and continuity of functions. It was proposed by Bledsoe [Ble90] as a challenge for automated theorem proving, because they are relatively easy for students of mathematics in the second year, however, extremely hard for automated reasoning: some problems remained open even until today. The proofs typically involve ϵ-δ arguments and are interesting because both logic and computation, including inequality reasoning and factorizations have to be combined to find a solution to the given problem. There exist several attempts in the literature that deal with the automation of this domain (see for example [MS99b, Bee98, YBG$^+$94, ST89, RL07, Mel98a, Mel98b, Hin94]. While it has already been shown in [Mei03] that proof planning is in principle capable to solve many problems of the limit domain which are out of the scope of traditional theorem provers, the contribution of this chapter is to show that the mathematical knowledge can be concisely represented in the proof document using the strategy language presented in Chapter 10.

14.1 Abstract Formalization

Instead of constructing the real numbers from scratch, we follow a more pragmatic approach and directly state the properties which uniquely define the real numbers as axioms. We will also provide a facility to perform explicit calculations with numerical constants, which is needed for many applications.

Definition 14.1.1 (Theory $T_{\mathbb{R}}$). *The theory of reals $T_{\mathbb{R}}$ is given by the following signature:*

$$+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \qquad (\textit{addition}) \tag{14.1}$$

$$\cdot \mathbb{R} \times \mathbb{R} \to \mathbb{R} \qquad (\textit{multiplication}) \tag{14.2}$$

$$-- : \mathbb{R} \to \mathbb{R} \qquad (\textit{addition-inv}) \tag{14.3}$$

$$^{-1} : \mathbb{R}\backslash\{0\} \to \mathbb{R} \qquad (\textit{multiplication-inv}) \tag{14.4}$$

$$< : \mathbb{R} \times \mathbb{R} \to \mathbb{B} \qquad (\textit{less}) \tag{14.5}$$

together with the following properties:

$$1 \neq 0 \tag{14.6}$$
$$\forall x, y. x + y = y + x \tag{14.7}$$
$$\forall x, y, z. x + (y + z) = (x + y) + z \tag{14.8}$$
$$\forall x. 0 + x = x \tag{14.9}$$
$$\forall x. (- - x) + x = 0 \tag{14.10}$$
$$\forall x, y. xy = yx \tag{14.11}$$
$$\forall x, y, z. x(yz) = (xy)z \tag{14.12}$$
$$\forall x. 1x = x \tag{14.13}$$
$$\forall x. x \neq 0 \Rightarrow x^{-1}x = 1 \tag{14.14}$$
$$\forall x, y, z. x(y + z) = xy + xz \tag{14.15}$$
$$\forall x, y. x = y \vee x < y \vee y < x \tag{14.16}$$

$$\forall x. x \not< x \tag{14.17}$$
$$\forall y, z. y < z \Rightarrow \forall x. x + y < x + z \tag{14.18}$$
$$\forall x, y. 0 < y \wedge 0 < y \Rightarrow 0 < xy \tag{14.19}$$

as well as the supremum property

$$\begin{aligned}\forall S. (\exists x. x \in S) \wedge (\exists M. \forall x \in S. x \leq M \\ \Rightarrow \exists m. (\forall x \in S. x \leq m) \wedge (\forall m'. \forall x \in S. x \leq m') \Rightarrow m \leq m'\end{aligned} \tag{14.20}$$

The supremum property states that any nonempty set of reals that is bounded above has a least upper bound. It is this is the property which distinguishes the reals from the rationals and makes the reals a complete metric space, meaning that every Cauchy sequence of points in $\mathbb{R}$ has a limit that is also in $\mathbb{R}$.

14.1.1 Performing Calculations

One major efficiency problem in the domain of the reals, but also in the domain of integers and naturals, is the computation with actual numbers. While the above standard formalization does only contain the numbers 0 and 1, it is convenient to extend the signature by numeric constants such that a term $x + x + x$ can be abbreviated to $3 \cdot x$. One possibility is the use of the successor function, as in Peano arithmetic, shown in Listing 17. However, this representation has the disadvantage that the time for the computation of $n * n$ grows exponentially, as shown in Figure 14.1[1]. As computations are often needed as intermediate steps, such an approach is not feasible. We therefore present two approaches to make such computation more efficient. In the first variant, we make use of the facilities provided by the strategy language and link the programming language to do the computation. This results in a semi-formal theory: All results will only be checked modulo the correctness of the computation of the programming language. The second variant shows a fully formal way, based on the idea of a binary number system as the representation of the numbers.

[1]The up- and downturns in the figure are due to garbage collection

```
inference addzero x+0 {*} == x
inference addsuc x+suc(y) {*} == suc(x+y)

inference mulzero x*0 {*} == 0
inference mulsuc x*suc(y) {*} == x+x*y

strategy simplify
  repeat rewrite * using addzero, addsuc, mulzero, mulsuc
```

Listing 17: Formalization of Peano arithmetic

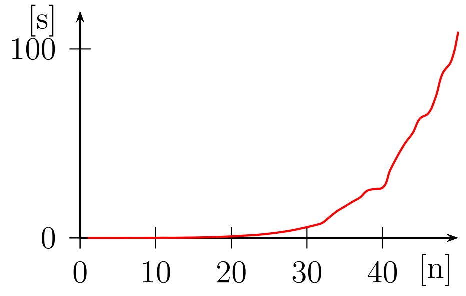


Figure 14.1: Time spent for computing $n * n$ in Peano arithmetic

Computations Inside the Programming Language

Listing 18 shows the semi-formal theory which uses the programming language to perform the numerical computation. Numbers are represented by a function `LINT`, which takes a single number as argument. The constant `LINT` is not needed, but acts as a marker to indicate that an integer number is following, which simplifies the matching process considerably. The resulting signature is infinite, as each number is represented as a separate symbol. The beauty of this formalization is its simplicity: (i) The connection to the programming language is very simple. (ii) After having specified the inferences, the full power of the strategy framework is available. Such a formalization drastically reduces the formalization costs and allows the rapid prototyping of the underlying concepts. Moreover, it is sufficient within the domain of proof tutoring where a reduction to basic principles is not needed.

```
inference addnum LINT(N) + LINT(M) {*}== LINT(O); {with O=(+ "N" "M")}
inference subnum LINT(N) + LINT(M) {*}== LINT(O); {with O=(- "N" "M")}
inference mulnum LINT(N) * LINT(M) {*}== LINT(O); {with O=(* "N" "M")}

strategy simplify
repeat
  rewrite * using addnum, subnum, mulnum
```

Listing 18: Semi-formal computations: calculations inside the underlying programming language

As a result, the time spent for numerical computations of even very big integers, such as $10^{100} * 10^{100}$ less $0.0001s$.

Binary Arithmetic

The general idea of this approach is to use a binary representation of numbers instead of the successor representation. As an advantage, the size of the representation is no longer linear in the size of the number. As this representation is not very human-readable, it is usually hidden from the user by (i) a parser that parses numbers in the decimal system and converts them into the binary system, (ii) a pretty printer which prints a number in the binary system as decimal number. The approach is sufficiently fast to deal with large numbers, and the resulting signature is finite.

To support binary arithmetic, we provide the following signature: $\Sigma = \{\overline{0}, \overline{1}, suc, 0\}$, where $0 : nat$, $\overline{0} : nat \to nat$ represent the bit zero[2], $\overline{1} : nat \to nat$ represents the bit one within a binary number, and $suc : nat \to nat$ represents the carry bit, which is used during computations only. Using this representation, each binary number is represented within the formal system by writing it from right to left. Prohibiting leading zeros gives a unique representation.

Example 14.1.2. *Thus the number* $37 = 2^5 + 2^2 + 2^0$ *is expressed as*

$$\overline{1}(\overline{0}(\overline{1}(\overline{0}(\overline{0}(\overline{1}(0)))))) \tag{14.21}$$

As learned in school, two numbers can be added by writing them down one below the other, and add them together starting from the right. This gives us the following rules,

$$0 + 0 \to 0 \tag{14.22}$$
$$0 + 1 \to 1 \tag{14.23}$$
$$1 + 0 \to 1 \tag{14.24}$$
$$1 + 1 \to 0, carry\ 1 \tag{14.25}$$

which are formalized as follows:

$$\overline{0}(m) + \overline{0}(n) \to \overline{0}(m + n) \tag{14.26}$$
$$\overline{0}(m) + \overline{1}(n) \to \overline{1}(m + n) \tag{14.27}$$
$$\overline{1}(m) + \overline{0}(n) \to \overline{1}(m + n) \tag{14.28}$$
$$\overline{1}(m) + \overline{1}(n) \to \overline{0}(suc(m + n) \tag{14.29}$$
$$suc(0) \to \overline{1}(0) \tag{14.30}$$
$$suc(\overline{0}(m)) \to \overline{1}(m) \tag{14.31}$$
$$suc(\overline{1}(m)) \to \overline{0}(suc(m)) \tag{14.32}$$

Similarly, multiplication and subtraction of positive numbers can be defined in an easy way (see Listing 19 for the formalization). The extension to integers is standard except the case for binary subtraction, where one needs the information which number has a greater magnitude.

Discussion

We must be able to perform simple computations, such as $2 + 5 = 7$, which naturally arise as subtasks in many applications. While most approaches rely on rewriting, the concrete representation is essential to make such computations efficient. An alternative

[2]The two variants correspond to the two cases that a zero is the rightmost number and the case that a zero is "in the middle" of a number

```
inference suczero suc(0){*}== BIT1(0)
inference sucbit0 suc(BIT0(m)){*}== BIT1(m)
inference sucbit1 suc(BIT1(m)){*}== BIT0(suc(m))

inference addzerol 0 + m{*}== m
inference addzeror m + 0{*}== m

inference addbit0bit0 BIT0(m)+BIT0(n){*}== BIT0(m+n)
inference addbit0bit1 BIT0(m)+BIT1(n){*}== BIT1(m+n)
inference addbit1bit0 BIT1(m)+BIT0(n){*}== BIT1(m+n)
inference addbit1bit1 BIT1(m)+BIT1(n){*}== BIT0(suc(m+n))

inference multzeror m * 0{*}== 0
inference multzerol 0 * m{*}== 0
inference multbit0bit0 BIT0(m) * BIT0(n){*}== BIT0 (BIT0 (m * n))
inference multbit0bit1 BIT0(m) * BIT1(n){*}== BIT0(m)+BIT0(BIT0(m * n))
inference multbit1bit0 BIT1(m) * BIT0(n){*}== BIT0(n)+BIT0(BIT0(m * n))
inference multbit1bit1 BIT1(m) * BIT1(n){*}== BIT1(m)+BIT0(n)+BIT0(BIT0(m * n))

inference elimleadingzero BIT0(0){*}== 0

strategy addbinary
  rewrite * using sucbin, addzerol, addzeror, addbit0bit0, addbit0bit1,
addbit1bit0, addbit1bit1

strategy multbinary
  rewrite * using multzeror, multzerol, multbit0bit0, multbit0bit1,
multbit1bit0, multbit1bit1, elimleadingzero

strategy simplify
  repeat rewrite * using addbinary, multbinary
```

Listing 19: Formalization of binary arithmetic

is to work in a semi-formal setting and to do the computation with the programming language. In addition to efficiency during the proof development, no time consuming theory development is necessary. Moreover, the computation can easily be expanded later and verified.

In the previous ΩMEGA system, numbers were formalized using the successor representation, which is clearly enough for problems dealing only with small numbers or no numbers at all. By using *annotated constants* and *annotated functions* it was possible to switch to an intermediate representation layer between the intuitive mathematical vernacular and the formal system [PS02, PS06]. While this is similar to our approach, the definition of annotated constants and functions working on them required deep knowledge of the implementation of the underlying prover. Our strategy language provides a clean interface between semi-formal representations and a fully formal representation. Moreover, following the document-centric approach, the operations can be specified within the document. By specifying computations as rewrite rules, the full power of the strategy framework is directly available.

The use of binary numbers can be seen to be standard in mathematical assistance systems. It was introduced by Harrison [Har98]. An alternative way consists of sep-

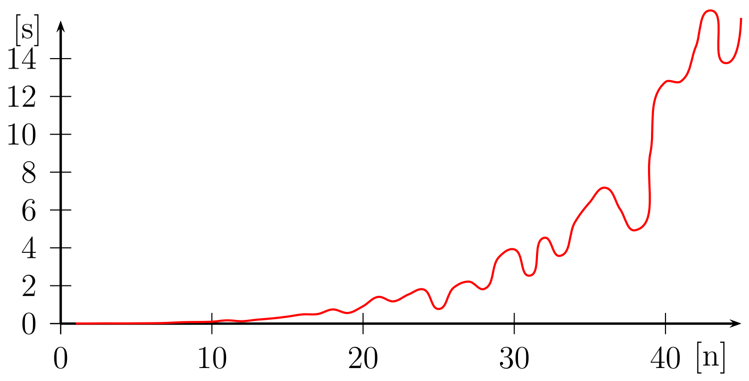


Figure 14.2: Time spent for computing $10^n * 10^n$ in binary arithmetic

arating deduction and computation (at least the decidable part), as done in *deduction modulo* [DHK03]. The idea is to define a congruence on propositions, such that $2 + 4$ is congruent to 6 and to use the congruence relation within deductions, such that for example reflexivity can directly be used to close the goal $\vdash 2 + 4 = 6$.

In the context of algebraic specifications there is the notion of *specification morphisms*, which are signature morphisms between particular specifications such that the reduct of each model of the target specification is a model of the source specification. Using specification morphisms, it is possible to define a theory of natural numbers using the successor representation and a theory of binary numbers and to relate them such that the representations are exchangeable.

14.1.2 Integration of Computer Algebra Systems

Computer algebra systems (CAS) and theorem provers support practical mathematical reasoning on a computer: While the former focuses on efficient representations and algorithms to perform algebraic computation, the latter focuses on rigorous logical validation, that is deduction. It is therefore important to let systems interact, either by importing deductions into CAS or by importing algebraic computation into theorem provers and verify them – this is often much easier than doing the computation itself.

Within recent years, the integration of CAS into theorem provers or vice versa has intensely been studied by several research groups. For example, computer algebra systems have been integrated in Coq [DM05], Hol [HT93a, HT93b], Isabelle [BHC95], HOL Light or ΩMega [KKS98]. On the other hand, theorem provers have been made available inside or defined on top of a CAS. For example, Pvs can be accessed from Maple [ADG+01], Theorema [BJD98, BJK+97] and Analytica [BCZ98] are defined within Mathematica. A comprehensive description of the different possibilities of connections is given in [Hom97, HC96]. Within a wider scope, OpenMath [CC99b, CC99a] provides a general framework to connect different systems together. Similarly, the Omrs project [GBC98] was initially developed as a framework for combining theorem provers and later extended to combine theorem provers and computer algebra systems.

We follow the first approach, namely to import the computation of a CAS into a theorem prover. In [HT93a], Harrison identifies three possible integration modes (i) to completely trust the CAS, (ii) to trust the CAS partially, and (iii) to not trust the CAS at all. The differences between these integration modes are as follows: When trusting the CAS, the computation is integrated in the prover and not verified. Partially trusting the CAS consists in accepting the result of the CAS during the interactive proof of the proposition, but checking the results of the CAS when exporting the theorem to the library. Finally, the last approach consists of using the CAS as oracle and to directly verify the result inside the prover. Note that this remains beneficial provided that checking the solution is much simpler than finding the solution. This is for example the case for computing factorizations of an expression: Once the factorization has been computed, checking whether the factorization equals the original term can easily be decided. We explore this problem more precisely in the sequel.

14.1.3 A Decision Procedure for the Equality of Polynomials over Rings

We often face the problem to show the equality between two terms t_1 and t_2, where each t_i is built from variables and constants by application of $+$ and $\cdot$. For example, when using a CAS to compute a factorization of t_1, resulting in t_2, we have to verify $t_1 = t_2$ as we do not want to trust the CAS. To deal with such tasks, we present in the sequel an algorithm to decide such an equality. The general idea is to reduce each t_i to a unique normal form, relying only on the ring axioms. If and only if both t_i have the same normal form, then they are equal.

Considering the equality axioms of a ring, the main problem for the normalization is the fact that the theory contains equations which cannot be ordered into a rewrite rule, such as the commutativity rule of the addition operator $+$ and $*$. Several approaches have been proposed in the literature to deal with that problem, such as special unification algorithms [JK86, Sie89] or unfailing completion procedures [BDP89]. We follow the approach taken in [MN90], called *ordered rewriting*. That is, rewrite rules which cannot generally be ordered are only allowed to be used if the concrete instance can be ordered. Thus, for the problematic equations, the problem of ordering the equations is moved from compile time to runtime.

Before defining a strategy deciding the equality for such expressions, we concretize the definitions needed within this chapter.

Definition 14.1.3 (Polynomial Expression)**.** *Let $x_1, \dots, x_n \in \mathbb{V}$ be variables and R be a ring. The set of polynomial expressions over $x_1, \dots x_n$ and the ring R, denoted by $R[x_1, \dots, x_n]$ is the smallest set satisfying the following conditions:*

- $x_i \in R[x_1, \dots, x_n]$
- *If $t \in R$ then $t \in R[x_1, \dots, x_n]$*
- *If $p_1, p_2 \in R[x_1, \dots, x_n]$ and $c \in R$ then*
  - $p_1 + p_2 \in R[x_1, \dots, x_n]$ *and*
  - $p_1 \cdot p_2 \in R[x_1, \dots, x_n]$
  - $c \cdot p_1 \in R[x_1, \dots, x_n]$

For *univariate polynomial expressions*, i.e., expressions that involve only one variable, the normal form is obtained by expanding the polynomial and removing superfluous terms such as $x - x$, resulting in a sum of products. The sum is then reordered and factors which differ only in a constant are merged, resulting in the following linear combination

$$p = \sum_{i=1}^{n} c_i x_i^i \tag{14.33}$$

For *multivariate polynomial expressions*, which involve several variables, there are several possible normal forms. The probably simplest is obtained by imposing an ordering $\prec$ on the variables and sorting the factors accordingly.

Definition 14.1.4 (Monomial, Normal Form)**.** *Let R be a ring. A* monomial *is an expression of the form*

$$c\prod_{i=1}^{n} x_i^{e_i} \tag{14.34}$$

where $c \in R\backslash\{0\}$ is called coefficient*. Given a monomial m and an ordering $\prec$ on $x_1, \ldots, x_n$, m is called in normal form if $x_i \prec x_j$ for $i < j$.*

Definition 14.1.5 (Normal Form Polynomial Expressions)**.** *Let $t \in R[x_1, \ldots, x_n]$ be a polynomial expression and $\prec$ be an ordering on x_i. Then t is called to be in* normal form *if it has the following form*

$$p = \sum_{i=1}^{k} c_i \prod_{j=1}^{n} x_j^{e_{i,j}} \tag{14.35}$$

where

- *each monomial is normal*
- $\prod_{j=1}^{n} x_j^{e_{i,j}} \hat{\prec} \prod_{j=1}^{n} x_j^{e_{k,j}}$ *for $i < k$, where $\hat{\prec}$ is an ordering over monomials induced by $\prec$.*

In the sequel, we use the lexicographic ordering induced by $\prec$ for $\hat{\prec}$.

Example 14.1.6. *The normal form of the polynomial expression $x(4+3(x-y+2))$ under the variable ordering $x \prec y$ is $3 * x^2 + 10x - 3xy$.*

The computation of a normal form can be performed automatically based on the above observations. The procedure involves the following steps:

1. Multiplying out factors, moving unary minus to the top of the monomials and removing binary minus
2. Sorting of the individual monomials
3. Rearranging and collecting monomials in the overall expression

Step 1: Factor out and basic simplification

The following rewrite system is terminating and confluent (see [Geh94]):

$$x + 0 \to x \qquad x + (-x) \to 0 \tag{14.36}$$
$$-(0) \to 0 \qquad -(-z) \to z \tag{14.37}$$
$$-(x+y) \to -(x) + -(y) \qquad x * (y + z) \to (x * y) + (x * z) \tag{14.38}$$
$$x * 1 \to x \qquad (x + y) * z \to x * z + y * z \tag{14.39}$$
$$x * 0 \to 0 \qquad x * (-z) \to -(x * z) \tag{14.40}$$

It multiplies out all factors, moves the $-$ sign to the front of the monomial, and performs trivial simplifications. Thus, we obtain as a result a sum of unordered monomials. It remains to order the sum and to collect monomials which differ only in their coefficients. This is done in step 2:

Example 14.1.7. *Performing step 1 of our decision procedure on $x(4 + 3(x - y + 2))$ results in the polynomial $x \cdot 4 + x \cdot 3 \cdot x + -x \cdot 3 \cdot y + x \cdot 3 \cdot 2$*

Step 2: Building Monomials

The goal of this step is twofold: (i) Sorting all numeric constants to the top and combining them, and (ii) sorting the variables according to $\prec$. We perform each step separately. The first step is realized by the rewrite system for multiplication of numbers enhanced by the following rewrite rules:

$$(x * y) * z \to x * (y * z) \tag{14.41}$$
$$x * y \to y * x \qquad \text{if } y \in \mathbb{Z} \text{ and } y \in \mathbb{V} \tag{14.42}$$
$$z * (x * y) \to x * (z * y) \qquad \text{if } x \in \mathbb{Z} \text{ and } y \in \mathbb{V} \tag{14.43}$$

The sorting is performed by the following rewrite system

$$(x * y) * z \to x * (y * z) \tag{14.44}$$
$$x * y \to y * x \qquad \text{if } x, y \in \mathbb{V} \text{ and } y \prec x \tag{14.45}$$
$$z * (x * y) \to x * (z * y) \qquad \text{if } x, y \in \mathbb{V} \text{ and } x \prec z \tag{14.46}$$

As a result, the monomials have the form $c_i \prod_{j=1}^{n} x_j^{e_{i,j}}$ or $\prod_{j=1}^{n} x_j^{e_{i,j}}$, that is, they do not necessarily have a numeric constant at the top. This is checked in a final step, in which 1 is added in front of a monomial if necessary. Both rewrite systems are confluent and terminating.

Example 14.1.8. *Applying step 2 of our decision procedure on the result of step 1, that is, $x \cdot 4 + x \cdot 3 \cdot x + -x \cdot 3 \cdot y + x3 \cdot 2$, results in $4x + 3xx + (-3)xy + 6x$.*

Step 3: Sorting and Collecting

Finally, we have to sort and collect the monomials. This is done by the following rewrite system:

$$(x + y) + z \to x + (y + z) \tag{14.47}$$
$$(c_1 * m_1) + (c_2 * m_2) \to (c_2 * m_2) + c_1 * m_1 \qquad \text{if } m_2 < m_1 \tag{14.48}$$
$$(c_1 * m_1) + (c_2 * m_2 + r) = c_2 * m_2 + (c_1 * m_1 + R) \qquad \text{if } m_2 < m_1 \tag{14.49}$$
$$c_1 * m + c_2 * m = (c_1 + c_2) * m \tag{14.50}$$

Example 14.1.9. *Performing step 3 on $4x + 3xx + (-3)xy + 6x$ yields the final result $10x + 3xx - 3xy$.*

As before, the corresponding rewrite systems can naturally be formalized in our strategy language. Moreover, once a decision procedure for polynomials over rings has been designed, a natural step consists of extending this procedure to polynomial expressions over fields, that is, to handle multiplicative inverses. This is done by eliminating all inverses (by using (14.14)) and then invoking the decision procedure for rings. More precisely, the extension works as follows:

- Perform the simplification step from the decision procedure for rings
- Move the inverses to the top of the monomials and collect them together.
- Select an inverse x^{-1}, multiply both sides of the equation with x, and perform the simplification. Repeat until there are no more inverses.

Discussion

In the previous ΩMEGA system, the SAPPER module was used to translate the computation of a computer algebra system to a sequence of tactic expressions that eventually verified the computation inside ΩMEGA's logic [Sor00]. To that end, SAPPER required the CAS to deliver a trace of its computation. As no state-of-the art CAS provided such a trace, we implemented our own computer algebra system μCAS. Similar to the approach mentioned above, we also verify the computation performed by the CAS; however, we do not require a trace of the computation and can therefore access any CAS without further work. Instead, we provide a decision procedure which can easily be implemented by relying on our strategy framework. The decision procedure can not only verify the computation of the CAS, but also any other equality over rings.

A similar approach is taken in COQ, which provides the tactic Ring to decide equality over rings (see [DM05] and [CfW04] for details). In contrast to our approach, the tactic Ring is implemented using the reflection principle: Instead of working on concrete terms, i.e., on terms of type $\mathbb{R}$, terms are translated into a new (inductive) type for polynomial expressions $\mathbb{P}$, on which a simplification function S is defined. Moreover, a function $[\![\cdot]\!] : \mathbb{P} \to \mathbb{R}$ is defined to translate polynomials back to their original form. Establishing the theorem $\forall p \in \mathbb{P}.[\![S(p)]\!] = [\![t]\!]$ then allows the use of S within the concrete representation. Similarly, Harrison represents polynomials (for one variable) as lists and defines a translation function from general algebraic expressions to polynomials and uses this form to perform arithmetic on polynomials.

14.2 Strategies of the Limit Domain

To tackle the problems of the limit domain a variety of heuristics have been developed within the last decades. In this section, we discuss several of these strategies and show how they can be easily formulated by means of the strategy framework developed in Chapter 10 and show how the Lim+ problem can be proved automatically.

The two important strategies, namely ComplexEstimate (see Listing 13 on page 206) and Factorbound (see Listing 15 on page 206) have already been discussed and presented in Chapter 10; we avoid a repetition here. Subsequently, we discuss the construction of instances for meta-variables, as well as the strategy LinearBound, which is used as a substrategy of the strategy Factorbound. Finally, we discuss a strategy Extract that extracts a subformula of an assumption.

14.2.1 Constructing Instances

The idea of the construction method is to construct an instance for a meta-variable that satisfies a set of constraints. Within the domain of the reals, we have to deal with a continuous variable domain and non-linear constraints, which are difficult to solve in general. However, it is often possible to reduce such non-linear constraints to linear constraints based on heuristics.

Let us point out that within our setting, the ability to check the satisfiability of a set of constraints is not sufficient: We have to come back with an instantiation and show that the constructed object indeed satisfies all its requirements. Moreover, we want to be able to generate a human-readable proof where the constructed objects are similar to those that can be found in textbooks. This usually excludes generic decision procedures because of the following reasons:

- They are based on a fixed theory, e.g., linear real arithmetic (without any user defined function symbols, such as $\sqrt{\cdot}$, min, cos, sin). This often results in large conditional output representations which can be decided, but not directly be used for instantiations. As an example, consider the solution of $x \leq z, y \leq z$ for z, which gives $(x \leq y \wedge z \geq y) \vee (x > y \wedge z \geq x)$ or $(x > 0 \wedge y > 0 \wedge z = \epsilon_1)$, depending on the underlying implementation. Here, ϵ_1 is a constant provided by the system that denotes a positive value that is infinitesimal wrt. the considered field.

- Most methods are iterative methods that enlarge the constraint set at each step in a mechanical way which is difficult to understand by humans because of their size. Thus, these methods do not provide an explanation of the solution.

In contrast, we want to support user-defined functions and be able to extend the capabilities of the construction method during the formalization of the theory. While there is in general no generic method to realize the construction of such an object, we subsequently present a heuristic method that works sufficiently well within the considered domain. The underlying idea is to incrementally restrict the domain of instances within the proof search, but not to commit a decision until all branches involving a specific meta-variable can be closed simultaneously. In other words: We follow a delayed closure approach. A similar approach was taken for proof planning in the previous ΩMEGA system, where the constraint solver COSIE (see [ZM04] for an overview) was used to compute answer constraints. Let us point out the main differences with respect to our approach:

- Constraints are not automatically passed to the constraint solver, but obtained by a query on the open goals. Moreover, constraint solving is directly implemented within the strategy language instead of an external program. This has the following advantages:

  (i) Constraint reasoning is not necessarily global, but can also be local with respect to a branch or a specific variable.

  (ii) There is no need to synchronize the constraint solver in the case of backtracking, which was necessary within the old architecture.

  (iii) Constraint passing is not encoded in the proof plan, therefore, the resulting plans are shorter and can directly be presented to the user.

  (iv) The constraint language as well as the simplification procedures can be extended and adapted during the theory development.

Simplification of Goals based on Assumptions

In addition to simplify goals using ComplexEstimate, we make use of the transitivity of $<$, which is expressed by the following theorem

$$a < b \wedge b \leq c \Rightarrow a < c \qquad (14.51)$$

in which we require both one assumption $a < b$ and the goal $a < c$ to be instantiated, with the hope that the resulting new goal $b \leq c$ is simpler. If desired, we can employ a meta-level check to compare the complexity of the involved terms, e.g., by a function `simpler` that compares the number of the involved function symbols. A possible realization is shown in Listing 20. Similar laws hold for $>$ and $\geq$. However, for efficiency it is reasonable to use either $<$ or $>$, as $a < b$ is equivalent to $b > a$.

```
inference solveb a<b{*}; b ≤ c ==> a<c{*};{where (simpler "b"
"a")}
inference solveb a≤b{*}; b ≤ c ==> a≤c{*};{where (simpler "b"
"a")}
inference solveb a<b; b ≤ c{*} ==> a<c{*};{where (simpler "b"
"a")}
inference solveb a≤b; b ≤ c{*} ==> a≤c{*};{where (simpler "b"
"a")}
```

Listing 20: Simplification of goals during proof search

In the case of the Lim+ problem, the inference is for example used to reduce the goal $|x-a| < \delta_1$ to $\delta \leq \delta_1$ based on the assumption $|x-a| < \delta$. It is important to note that one should give such a simplification higher preference than the axiom rule, which would also be applicable and instantiate δ with δ_1.

Finding Linear Bounds

It is often necessary to find linear upper and lower bounds for limit problems. To that end, we design a strategy LinearBound, which was already used as a substrategy of the strategy FactorBound. LinearBound is based on heuristics and aims at generating simple constraints that can easily be solved. The strategy is based on the triangle inequality, that is,

$$|x+y| \leq |x| + |y| \tag{14.52}$$
$$|x+y| \geq |y| - |x| \tag{14.53}$$

The trick of the strategy is that the resulting condition is linear and can therefore easily be processed. Moreover, this increases the chances that the answer constraint is compact and simple.

Suppose we have an assumption $|x-3| < \boldsymbol{\delta}$ with the additional condition that $\boldsymbol{\delta} > 0$ (as usual for ϵ-δ proofs). In this case, we can easily compute a lower bound

$$|x-1| = |x-3+2| \overset{(14.52)}{\leq} |x-3| + 2 < \boldsymbol{\delta} + 2 \tag{14.54}$$

respectively an upper bound:

$$|x-1| = |x-3+2| \overset{(14.53)}{\geq} 2 - |x-3| \geq 2 - \boldsymbol{\delta} \tag{14.55}$$

The disadvantage is that the solution involves δ, i.e., subsequent transformations that are based on this estimation become dependant on δ. However, refining the constraint by imposing a further condition, e.g., $\boldsymbol{\delta} < 1$ allows us to get numerical values as bounds, e.g., $|x-1| < 3$ and $|x-1| > 1$, which can easily be used in subsequent computations. In particular, the additional condition can easily be collected when the min function is available. The resulting strategy is shown in Listing 21.

```
strategy linearbound
cases
  (abs(X-A))< N,* |- (abs(X-B))<=MV where
    (and (variable-metavar.is "N")
         (variable-metavar.is "MV"))
  with
    R = (maxima-simplify "B-A")
    FCONS = (maxima-simplify "1+abs(B-A)")
->
proof
  N < 1
  (abs(X-B)) = (abs(X-A+R))
  .<= (abs(X-A))+(abs(R))
  .<=1+(abs(R)) from L
  put MV = FCONS
qed
```

Listing 21: Instantiation strategy for finding upper bounds

Constraint Simplification

Given a set of goal constraints, constraint simplification works by applying conditional rewrite rules to the constraints with the goal to isolate meta-variables. Indeed, the simplification rules of Cosie (see [ZM04] p. 84f) can easily be modeled within our strategy language. Due to compilation, the resulting strategies are sufficiently efficient. However, as the constraints may be scattered among the different branches of the search tree, it is necessary to unify all goal constraints in a single conjunction and starting a new proof tree.

14.2.2 The Extraction Strategy

It is common to alternate between forward and backward reasoning during proof search: Work backwards as long as suitable inferences are available if backward reasoning is no longer possible, start to work forwards from the assumptions, aiming at closing the given goal. During the second phase, new goals might arise, for which backward reasoning is started again. In the context of the limit domain, a common situation is that a certain conditional assumption, e.g., an estimate that depends on certain conditions, is used to simplify the current goal. Strategic reasoning allows us to trigger the unwrapping of a condition on demand and solve the arising subgoals afterwards. Note that while it is also often possible to rely on the deep axiom rule, a separate extraction of an assumption often results in more readable proofs.

To support the extraction of a subformula of an assumption, we provide a strategy Extract that takes a task position to be extracted as input, and applies elimination rules to the formula until the subformula is extracted[3]. The strategy is provided as a library function.

Example 14.2.1. *Consider the task*

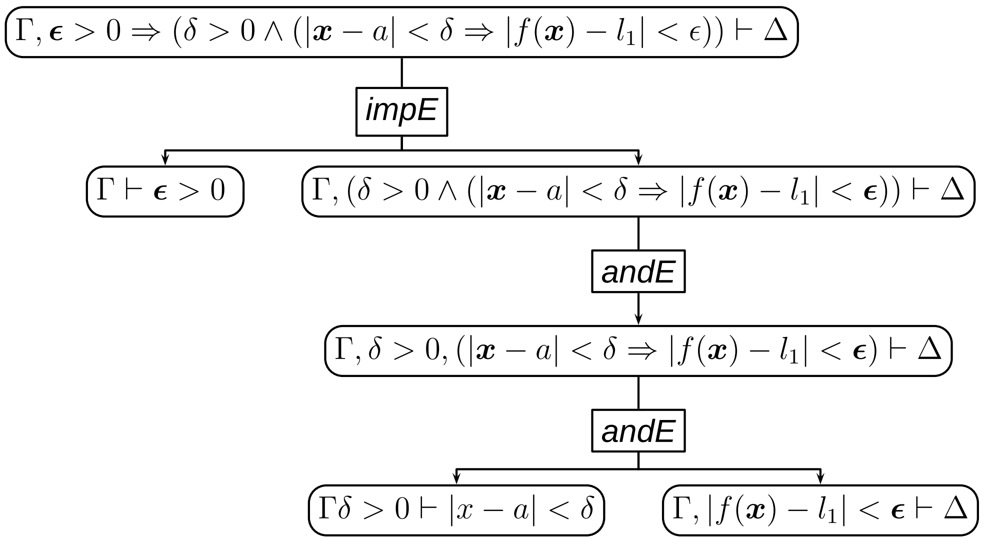
$$\Gamma, \boldsymbol{\epsilon} > 0 \Rightarrow (\delta > 0 \wedge (|\boldsymbol{x} - a| < \delta \Rightarrow |f(\boldsymbol{x}) - l_1| < \epsilon)) \vdash \Delta \tag{14.56}$$

[3]Within the Core calculus, this means that only the windows are adapted

Extraction of the subformula $|f(x) - l_1| < \epsilon$, i.e., applying elimination rules to unwrap the assumption $|f(x) - l_1| < \epsilon$ results in the following proof script:

```
proof
  L1: $\epsilon > 0$
  L2: consider $\delta$ such that $\delta > 0 \wedge (|x - a| < \delta \Rightarrow |f(x) - l_1| < \epsilon)$ from
L1
  L3: $\delta > 0$ from L2
  L4: $|x - a| < \delta \Rightarrow |f(x) - l_1| < \epsilon$
  L5: $|x - a| < \delta$
  L6: $|f(x) - l_1| < \epsilon$ from L4,L5
qed
```

which is exactly the result of translating the following proof tree


$\Gamma, \boldsymbol{\epsilon} > 0 \Rightarrow (\delta > 0 \wedge (|\boldsymbol{x} - a| < \delta \Rightarrow |f(\boldsymbol{x}) - l_1| < \epsilon)) \vdash \Delta$
impE
$\Gamma \vdash \boldsymbol{\epsilon} > 0$
$\Gamma, (\delta > 0 \wedge (|\boldsymbol{x} - a| < \delta \Rightarrow |f(\boldsymbol{x}) - l_1| < \boldsymbol{\epsilon})) \vdash \Delta$
andE
$\Gamma, \delta > 0, (|\boldsymbol{x} - a| < \delta \Rightarrow |f(\boldsymbol{x}) - l_1| < \boldsymbol{\epsilon}) \vdash \Delta$
andE
$\Gamma\delta > 0 \vdash |x - a| < \delta$
$\Gamma, |f(\boldsymbol{x}) - l_1| < \boldsymbol{\epsilon} \vdash \Delta$

14.2.3 Complex Estimate Revisited

In Chapter 10, we have already shown how to model the strategy ComplexEstimate within our strategy framework. Here, we present a more general version. As before, ComplexEstimate is applied to tasks whose goal is of the form $|b| < \epsilon$ and which have a support formula $|a| < \epsilon'$ where a occurs as linear combination in b. However, in contrast, the version below uses forward reasoning, which is better suited for proof presentation, and does not require the support formula $|a| < \epsilon$ to be at top-level. As a consequence, the unwrapping of the subformula is controlled directly inside ComplexEstimate, and only suitable formulas are extracted. The method is shown in Listing 22.

14.3 The Lim+ Problem

The so-called Lim+ problem is the problem to show that

$$\lim_{x \to a} f(x) = l_1, \lim_{x \to a} g(x) = l_2 \vdash \lim_{x \to a} f(x) + g(x) = l_1 + l_2 \qquad (14.57)$$

```
strategy ComplexEstimate
cases
  [$|a| < e_1$],* ⊢ $|b|$ < $\epsilon$
  where (not (terms-are.equal "k" "0"))
  with $\sigma$ in (findaddsubst "a" "b")
    (k,l) = (maxima-divide "b" (substapply $\sigma$
"a")) ->
  proof
    put $\sigma$
    apply extract taf([$|a| < e_1$])
    L2: $|k * a| < \frac{\epsilon}{2}$
    proof
      put $e_1 = \frac{\epsilon}{2k}$
      $|k| * |a| < |k| * e_1$
    qed
    L3: $|l| < \frac{\epsilon}{2}$
    L4: $|b| = |k * a + l|$
    L5: $. \leq |k * a| + |l|$
    L6: $. < \epsilon$
  qed
```

Listing 22: Improved Variant of ComplexEstimate

After expanding the definitions of lim, the proof situation looks as follows:

$$\begin{aligned}
T_1 :& \boldsymbol{\epsilon_1} > 0 \Rightarrow \delta_1 > 0 \wedge (|\boldsymbol{x_1} - a| < \delta_1 \Rightarrow \boxed{|f(\boldsymbol{x_1}) - l_1| < \boldsymbol{\epsilon_1}}),\\
& \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2})\\
& \epsilon > 0, |x - a| < \boldsymbol{\delta} \vdash \boxed{|f(x) + g(x) - (l_1 + l_2)| < \epsilon} \qquad (14.58)\\
T_2 :& \boldsymbol{\epsilon_1} > 0 \Rightarrow \delta_1 > 0 \wedge (|\boldsymbol{x_1} - a| < \delta_1 \Rightarrow |f(\boldsymbol{x_1}) - l_1| < \boldsymbol{\epsilon_1}),\\
& \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2}),\\
& \epsilon > 0 \vdash \boldsymbol{\delta} > 0
\end{aligned}$$

In this situation, the modified version of ComplexEstimate is applicable, using the substitution $\sigma = \{x_1 \mapsto x\}$, and the specified proof fragment is executed: The substitution σ is applied, and the formula $|f(x) - l_1| < \epsilon_1$ is extracted. The extraction results in the following new goals:

$$\begin{aligned}
T_3 :& \epsilon > 0\\
& |x - a| < \boldsymbol{\delta}\\
& \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2}) \vdash \boldsymbol{\epsilon_1} > 0 \qquad (14.59)\\
T_4 :& \delta_1 > 0,\\
& \epsilon > 0,\\
& |x - a| < \boldsymbol{\delta}\\
& \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2}) \vdash |x - a| < \delta_1 \qquad (14.60)
\end{aligned}$$

and an additional task T_5 on which the execution of ComplexEstimate continues. T_5 has the following form:

$$\begin{aligned}T_5 :& \delta_1 > 0,\\ & \epsilon > 0,\\ & |x - a| < \boldsymbol{\delta}\\ & \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2})\\ & |f(x) - l1| < \epsilon_1 \vdash |f(x) + g(x) - (l_1 + l_2)| < \epsilon \end{aligned} \tag{14.61}$$

ComplexEstimate now inserts the body of the specified proof script and applies the substitution $\epsilon_1 \mapsto \frac{\epsilon}{2}$. Moreover, it adds the new facts $L2$ to $L6$ as specified in the proof strategy, which can all directly be closed except of $L3$, which is expanded to the following task:

$$\begin{aligned}T_6 :& \delta_1 > 0,\\ & \epsilon > 0,\\ & |x - a| < \boldsymbol{\delta}\\ & \boldsymbol{\epsilon_2} > 0 \Rightarrow \delta_2 > 0 \wedge (|\boldsymbol{x_2} - a| < \delta_2 \Rightarrow |g(\boldsymbol{x_2}) - l_2| < \boldsymbol{\epsilon_2})\\ & |f(x) - l_1| < \epsilon_1 \vdash |g(x) - l_2| < \frac{\epsilon}{2} \end{aligned} \tag{14.62}$$

To solve the remaining tasks, the goals are simplified based on the simplification rules introduced above. T_6 is reduced to the new task $\epsilon_2 \leq \frac{\epsilon}{2}$, as well as in two conditions $\epsilon_2 > 0$ and $|x - a| < \delta_2$. Moreover, the goals $|x - a| < \delta_1$ and $|x - a| < \delta_2$ are reduced to $\delta \leq \delta_1$ respectively $\delta \leq \delta_2$. Solving the constraints in a new tree results in the conjunction

$$\delta \leq \delta_1 \wedge \epsilon_2 \leq \frac{\epsilon}{2} \wedge \epsilon_2 > 0 \wedge \delta \leq \delta_2 \tag{14.63}$$

which can be solved by constraint simplification and results in the instantiations $\epsilon_2 = \frac{\epsilon}{2}$ and $\delta = \min\{\delta_1, \delta_2\}$. The resulting proof script (showing only the most abstract granularity level) is shown in Figure 14.3.

While the proof script shown above is already readable, there is still some potential for improvement. In an actual textbook proof, the first two lines would probably not be presented. Moreover, the fact `L4` would only be derived once instead of twice, and both **assume** blocks would probably be combined. Finally, $1 * (f(x) - l_1)$ would be replaced by $f(x) - l_1$. These are minor improvements which can relatively easy be performed by transforming the generated script above or by incorporating the special cases in the strategy ComplexEstimate.

14.4 Comparison with Multi and Discussion

While it has already been shown that the limit domain – in particular the Lim+ problem – can be automated in principle based on proof planning with multiple strategies (see [Mei04]), our approach represents a considerable advancement of the techniques developed in the work cited above with respect to efficiency, the control flow needed to solve the problem, the specification costs for the design of the domain dependent knowledge, the quality of the resulting proof plan, and finally the possibility to evaluate the whole approach. Moreover, we have generalized many existing strategies, such as ComplexEstimate, as well as integrated new strategies, including a decision procedure that allows the

```
theorem th1: $\lim_{x\to a} f(x) = l_1 \wedge \lim_{x\to a} g(x) = l_2 \Rightarrow \lim_{x\to a} f(x) + g(x) = l_1 + l_2$
proof
  have L1: $\frac{\epsilon}{2} > 0 \Rightarrow \delta_1 > 0 \wedge (|x-a| < \delta_1 \Rightarrow |f(x) - l_1| < \frac{\epsilon}{2}$
  have L2: $\frac{\epsilon}{2} > 0 \Rightarrow \delta_2 > 0 \wedge |x-a| < \delta_2 \Rightarrow |g(x) - l_2| < \frac{\epsilon}{2}$
  assume L3: $\epsilon > 0$
    have L4: $\frac{\epsilon}{2} > 0$
    have L5: $\delta_1 > 0$
    have L6: $\frac{\epsilon}{2} > 0$
    have L7: $\delta_2 > 0$
  thus $\min\{\delta_1, \delta_2\} > 0$
  assume $\epsilon > 0$ and $|x-a| < \min\{\delta_1, \delta_2\}$
    have L8: $\frac{\epsilon}{2} > 0$
    consider $\delta_1$ such that L9: $\delta_1 > 0 \wedge (|x-a| < \delta_1 \Rightarrow |f(x) - l_1| < \epsilon)$ from L8
    have L10: $\delta_1 > 0$ from L9
    have L11: $|x-a| < \delta_1$
    have L12: $|f(x) - l_1| < \frac{\epsilon}{2}$ from L11
    have L13: $|1 * (f(x) - l_1)| < \frac{\epsilon}{2}$
    have L14: $|g(x) - l_2| < \frac{\epsilon}{2}$
    have L15: $|b| = |1 * f(x) - l_1 + g(x) - l_2|$
    have L16: $. \leq |1 * f(x) - l_1| + |g(x) - l_2|$
    have L17: $. < \epsilon$
  thus $|f(x) + g(x) - (l_1 + l_2)| < \epsilon$
qed
```

Figure 14.3: Proof script automatically generated for the lim+ problem

prover to close equality tasks over the reals automatically. This shows that it is relatively easy to encode the knowledge that was used in the proof planer MULTI in our strategy language. Due to large amount this knowledge in MULTI, only the major strategies were formalized.

- The new strategy language allows the combination of several strategies based on meta-level reasoning: If the assumptions contain an appropriate inequality that occurs as linear combination in the goal, then (1) extract the inequality (which is a variant of MULTI's strategy UnwrapHyp), before (2) using the extracted inequality in the goal. This has the significant advantage that the support formula is unwrapped only if it is effectively needed in the subsequent proof, rather than unwrapping "promising supports" without any motivation. Moreover, the control flow remains local to the overall strategy and need not to be temporarily stored (e.g., on a backboard or as an annotation). Eager instantiation of meta-variables during the extraction process can directly be controlled within the strategy without further meta-level reasoning, which simplifies the required control rules (such as MULTI's InstIfDetermined), and makes the interruption of strategies, e.g., to unwrap a formula or to propagate meta-variable instantiations, superfluous. Moreover, as the instantiation of only a subset of the constraints is delayed, the constraint set that needs to be managed becomes much smaller and therefore easier to solve.

  Overall, at the most abstract level, the number of strategic actions is reduced from twelve (see Figure 14.4) to UnfoldDefs, ComplexEstimate, Simplify, SolveContraints, that is, only to four actions. Inferences corresponding to the methods for unfolding the definitions of lim need no longer be specified manually. As the inferences already decompose the formula in the case of backward reasoning, the method NormalizeLineTask, which achieves this goal in MULTI, is no longer needed. Moreover, the methods TellCS-B that sends constraints to the constraint solver and the method Simplify-B that invokes a CAS to simplify a formula are no longer needed. At the level of control rules, the control rule choose-unwrap-support and eager-instantiate become superfluous. Let us point out that the new formalization consists only of several hundred lines of code compared to the 17000 before. While there is also control knowledge that is not necessary to solve the Lim+ problem, the reduction from 117 to 34 lines of code in the case of ComplexEstimate shows the significant benefit of the new strategy language.

- It becomes possible to evaluate the required techniques easily. For the Lim+ problem, these are:
  - ComplexEstimate to determine a useful inequality in the assumptions, techniques to unwrap it, and to rewrite the goal accordingly.
  - The transitivity of equality to simplify a goal inequality using an existing support, based on the underlying idea to postpone the instantiation of a meta-variable (the δ in the Lim+ proof) by simplifying it to linear inequalities.
  - A strategy to simplify/solve simple inequalities such as $\frac{\epsilon}{2} > 0$ based on $\epsilon > 0$, which can be accomplished by rewriting.
  - A domain specific strategy to simplify constraints (to combine the constraints $\delta \leq \delta_1$ and $\delta \leq \delta_2$).
- Proof plans encode mathematical proofs rather than proofs including control information. In particular, the methods to manage the constraint solving, such as TellCS-B, become superfluous. As a consequence, the resulting proof plan can easily be translated to a natural looking proof script.
- The constraint solver is specified by rewriting rules as a strategy. This has the advantage that the underlying knowledge is explicit and that no external constraint solver satisfying the requirements for proof planning needs to be implemented from scratch. Most importantly, the solver becomes extensible. This is highly valuable in the interactive setting, as a rational instantiation often requires user-defined functions, such as the min function above.

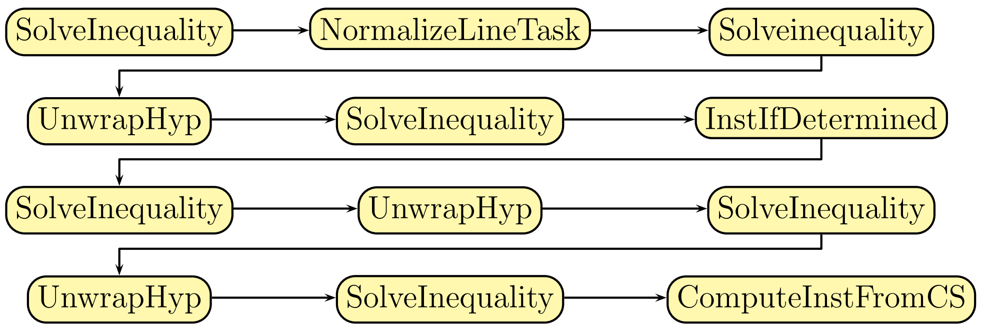


Figure 14.4: Strategic Control Flow in MULTI

14.5 Related Work

14.5.1 Bledsoe's Imply and STR+VE prover

One of the first attempts towards automating elementary mathematical analysis and set theory goes back to Bledsoe in the seventies and his theorem prover Imply [BBH72]. Bledsoe developed a strategy for proving inequalities [BH80] over the reals and introduced the so-called *limit-heuristic* to reduce inequality goals. The STR+VE prover [Hin94] proved the Lim+ problem and its variants. The original method ComplexEstimate is inspired by Bledsoe's limit heuristic. Our version further refines ComplexEstimate in the following ways: (1) Control of the extraction process of a suitable support formula; (2) Direct instantiation of some meta-variables, (3) Control of the shape of the resulting proof script.

14.5.2 Weierstrass

In [Bee98], Beeson describes techniques implemented in his prover Weierstrass to automatically find ϵ-δ proofs in elementary analysis. Weierstrass is based on the sequent calculus and performs a backward search to prove theorems. It uses meta-variables to delay the choice for existential witnesses. Simplification of expressions is performed by so-called *operations*, which are special algorithms to rewrite expressions to equal expressions under certain side conditions. Side conditions are either inferred, checked, or assumed.

In addition to the factor-bounding heuristic which was already described above, Weierstrass contains algorithms to find lower and upper bounds, a heuristic based on the mean value theorem to prove inequalities, and uses a list of known inequalities to simplify the goal, such as $|\sin x| < |x|$ for small x. Among others, Weierstrass can prove the continuity of x^3 and $\sqrt{x}$ on closed intervals, and the uniform continuity of $\sin(x)$ and $\cos(x)$.

While the proofs of the theorems produced by Weierstrass are presented in an abstract way and many interesting heuristics are implemented, there is no general language to express the reasoning techniques/heuristics used by the prover. Rather, they are coded in the underlying programming language of the prover and only partially described within research papers. Moreover, no detailed information about the formalization of the background theory is given, which makes a detailed comparison difficult.

14.5.3 An Interactive Calculus Theorem-prover for Continuity Properties

In [ST89] Suppes and Takahashi present an interactive calculus theorem prover for continuity properties, which is build on top of the CAS Reduce. Their focus is on the development of interactive methods of theorem proving which are practical for students to use with no programming background and without extensive mathematical background. The environment provided by the authors allows the construction of proofs with gaps, which are closed by a simple resolution theorem prover VE, or its special variant OE, which assumes properties of ordered fields to close the gap. Moreover, the authors provide a strategy M to prove simple algebraic theorems, such as $x * y = z \wedge y \neq 0 \Rightarrow x = \frac{z}{y}$. M essentially applies decomposition rules to formulas to get sequents containing only literals and uses then forward exploration to deduce facts from that literals. Functions such as $|\cdot|$, min and max are expanded using a list of theorems. Data and therefore also numbers are represented in Lisp without types and CAS REDUCE is used for simplification without justifying the results.

While their system did not allow the automatic construction of proofs for continuity properties, it represents on of the first attempts to develop a prover for that domain and to automate simple subtasks. From our perspective it is interesting to see the interactive proof construction as a proof plan, whose gaps are closed by a several strategies provided by the authors.

14.5.4 Theorema

The so-called PCS method (see [Buc01]) for analysis has been developed for the THEOREMA system. The PCS method reduces proving in elementary analysis, in particular formulas with alternating quantifiers on functions systematically to the solution of inequalities over the real numbers. The PCS method is able to solve the Lim+ problem, provided that appropriate lemmas are specified. Constraints are solved using Collins method for quantifier elimination, which has the disadvantage that no textbook-like instantiations are computed. Moreover, THEOREMA does not provide any functionality to specify control knowledge within the proof document: A *prover* is a fixed combination of inference rules, suitable for a particular class of proof situations (see [Dup00] for details), where fixed means that the inference rules as well as the order in which they are applied are fixed. The prover needs to be encoded in the programming language of MATHEMATICA.

14.5.5 Oyster Clam

Oyster Clam succeeded in proving the Lim+ property using colored rippling (see [YBG$^+$94, Bun96] for details). Compared to our approach, the rippling heuristic is more general as our strategies as it can also be applied to other domains. However, it requires a specific axiomatization of the problem and an annotation of the goal and targets of the rippling process. Moreover, the definitions were already unfolded in the original problem. Due to the use of a CAS, our proof strategy depends less on the formalization. In addition, we produce a readable declarative proof script.

14.6 Summary

In this chapter we showed how to express common patterns of reasoning of the limit domain using the strategy language developed in the Chapter 10 (Evaluation E2, Section 1.1). Compared with a reference implementation of the previous proof planner MULTI, the specification costs of proof strategies measured in lines of code are reduced and the readability and maintainability of the strategies increased. The possibility to specify proof strategies within the proof document increases the usability of interactive theorem provers and proof planners, but also provides a possibility to evaluate systems that are based on heuristics, as all knowledge becomes explicit.

15 Integration Into a Scientific Text-Editor

In this chapter, we describe the integration of ΩMEGA into the scientific text-editor TEXMACS (see [vdH01] for a description). TEXMACS provides professional type-setting and supports authoring with powerful macro definition facilities like those in LaTeX, but the user works on the final document ("What you see is what you get", WYSIWYG). The motivation was to enable the *document-centric approach* to formalizing and verifying mathematics, that is, the development of mathematical documents in a publishable style, which, in addition, are formally validated, resulting in *certified mathematical documents*. To realize this overall goal, several people of our research group worked together on solutions both for the proof assistant, as well as for the scientific text-editor. The development started with [ABFL06], and resulted in a first prototype, called PLATO [MW07]. The goal of this section is to give an overall overview of the developed architecture in the context of existing work and to discuss the changes in the architecture of the proof assistant to accomplish the requirements of the scenario.

15.1 Historical Remarks and Design Goals

Early interactive theorem provers, such as LCF, operated with the prompt of the underlying programming language based on a read-eval-print loop. Later, this interaction mode was slightly improved by the introduction of a separate, prover-specific command language in provers like ISABELLE or COQ. A major advance towards user interfaces was the development of the PROOFGENERAL interface [Asp00], which provides a prover independent communication protocol between a so-called *display*, and a *reasoner*, via a central component called *broker*. Within the display, the complete document is shown, which is split into two parts: an already processed part which cannot further be edited, and an editable region of unprocessed text. The pointer/focus that separates both regions can be moved forwards, triggering the processing of the next command, or backwards, invoking an undo operation within the prover. In this sense, the interface is still in the spirit of a sequential, command-line based interaction model with two essential commands next and backtrack. PROOFGENERAL has been ported to other editors/IDE-environments, such as Eclipse (see [ALW07] and [JC08]), or NetBeans (see [Gas09]).

Considering the existing work, our main design goal was to consider the input document as a first-class citizen with an internal state, without imposing any restrictions on how the document is structured, how it is edited, and which language is used in the document, at the expense of higher processing costs. More precisely, we support the following features:

- **Multi Focus Editing:** The single focus approach is rather restrictive as it forbids the user to edit parts of the document that have already been processed. Therefore, we support a *multi focus editing* approach, where the user can edit any part of the document. Backtracking and rechecking is automatically triggered by the proof assistant.

- **Unrestricted Input:** The input language of the document is independent of the syntax of the proof assistant. In particular, the user is not required to give justification hints of how to discharge proof obligations arising from intermediate steps. Moreover, we do not require the use of the language of the proof assistant within our proof document.

- **Incremental Proof Checking:** Proof checking without any hints can become complex and time consuming. Therefore, we rely on an incremental asynchronous approach: The editor never waits for the verifier and editing is therefore always possible. The prover checks the consistency of the document in the background without explicit requests from the user. The prover itself has full control to schedule the resulting proof scripts in any order, thereby exploiting parallelism.

- **Continuous Proof Checking:** We support the continuation of proof checking after the first encountered error. While this is a main feature of declarative proof languages, let us stress that this feature is not standard, e.g., it is not supported by ISABELLE.

- **Partial Proof Steps:** We support the processing of partial proof scripts.

Moreover, in addition to proof checking we provide additional services to support the generation of documents. These include:

- **Proof Script Generation:** Instead of only checking user-written proofs, we support the automatic generation of declarative proof scripts (e.g., by declarative strategies) which are inserted into the proof document.

- **Granularity Change:** One main feature of the $\mathcal{PDS}$ is the ability to maintain subproofs at different levels of granularity simultaneously, including a so-called *$\mathcal{PDS}$-view* representing a complete proof at a specific granularity. Given a proof step at a specific granularity, we support to switch to a higher/lower granularity on demand.

- **Proof Command Completion:** Given a partial cases construct, where some of the cases are already given, or a partial subgoals command, where some of the subgoals are given, we provide a facility to automatically complete the missing cases/subgoals and to patch them into the document.

- **Proof By Pointing:** The idea in proof by pointing [BKT94] is that the mere gesture of pointing at a subexpression in a subgoal is enough to synthesize appropriate commands for the system. We feature this approach by context sensitive menus that offer possible transformations at the position the user has clicked at.

15.2 Architecture and Communication

Figure 15.1 shows the overall architecture of our approach. It essentially consists of three components: The text-editor, which is in our case TEXMACS, the proof assistant ΩMEGA, and a mediator that communicates with both the text-editor and the proof assistant.

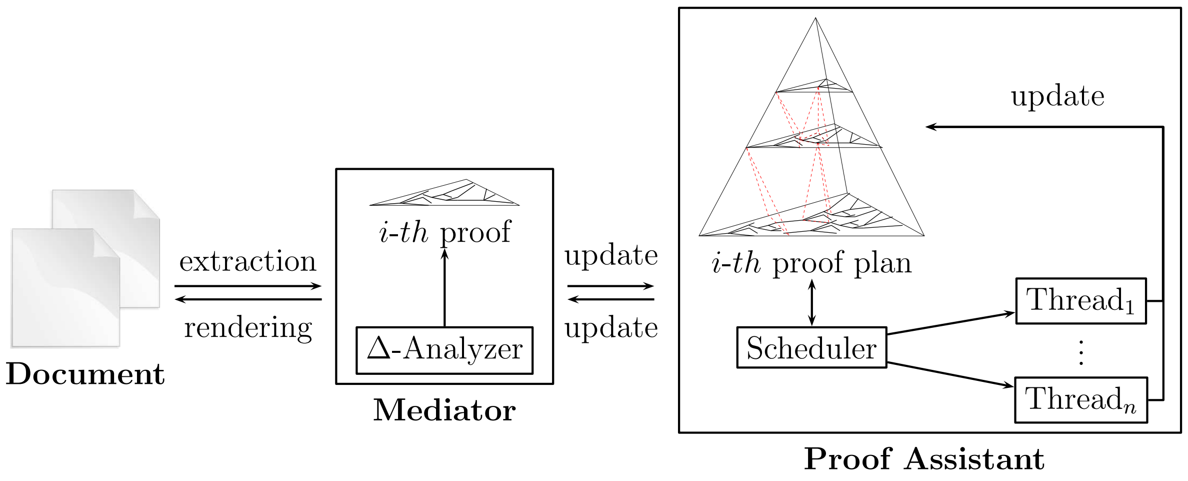


Figure 15.1: Architecture of the integration of a text-editor and a proof assistant via a mediator

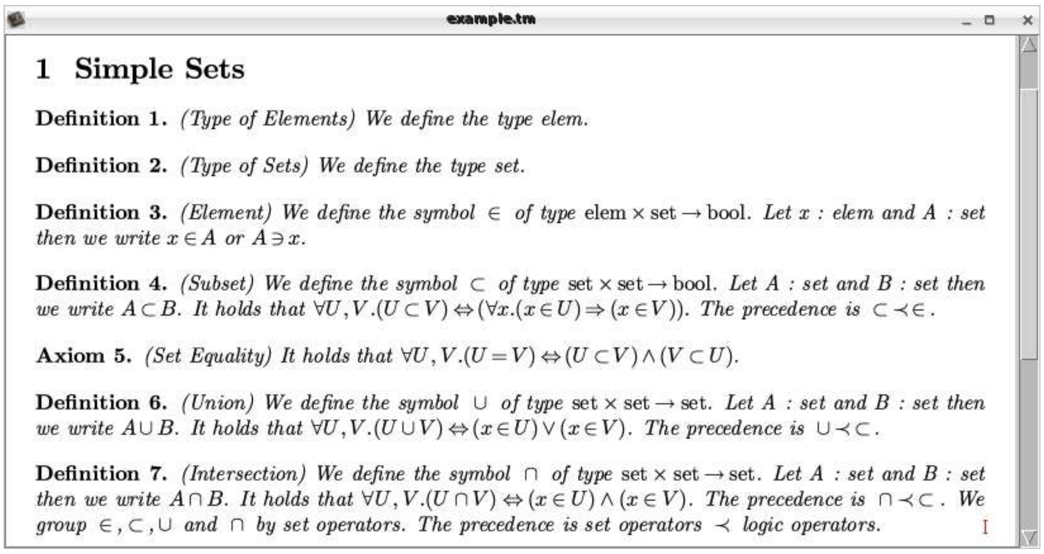
The roles of the components are:

Text-Editor: The text-editor is the front-end; it displays the mathematical document together with status information and provides facilities to edit the document.

Mediator: The role of the mediator is to abstract from the concrete implementation of the text-editor that is used to display the documents. It also abstracts from the concrete input language in which the background theory is formalized. Moreover, it must preserve the consistency between the text-editor and the proof assistance system. This is achieved by providing facilities to extract the formal content from the document and sending it to the proof assistant, as well as by translating new parts generated by the proof assistant back to the text-editor. Additionally, services and feedback of the proof assistance system are made available within the text-editor.

Proof Assistance System: The role of the proof assistance system is to maintain a formal version of the document and to provide additional services on top of the formal representation. In particular, this includes refinement of the extracted abstract proof sketches to machine-checkable proof objects, detection and reporting of errors, and automatic generation of human-readable proof scripts.

Let us now consider the interaction between these components by showing the initial workflow, which is as follows: The user opens a theory file (possibly empty), which is immediately uploaded by the text-editor and processed by the mediator. The mediator in turn extracts a formal document, which is sent to the reasoner, which starts to discharge proof obligations. Subsequently, the author changes the document either by writing new parts or by invoking automated reasoning procedures. As an example, Figure 15.2 shows an initial document about *Simple Sets* written in TEXMACS. This theory defines two base types and several set operators together with their axioms and notations. In general, theories are built on top of other theories and may contain definitions, notations, axioms, lemmas, theorems and proofs. Note that in the example set equality is written as an axiom because equality is already defined in the base theory.



1 Simple Sets

Definition 1. *(Type of Elements) We define the type elem.*

Definition 2. *(Type of Sets) We define the type set.*

Definition 3. *(Element) We define the symbol* $\in$ *of type* $\mathrm{elem} \times \mathrm{set} \rightarrow \mathrm{bool}$. *Let x : elem and A : set then we write $x \in A$ or $A \ni x$.*

Definition 4. *(Subset) We define the symbol* $\subset$ *of type* $\mathrm{set} \times \mathrm{set} \rightarrow \mathrm{bool}$. *Let A : set and B : set then we write $A \subset B$. It holds that $\forall U, V.(U \subset V) \Leftrightarrow (\forall x.(x \in U) \Rightarrow (x \in V))$. The precedence is $\subset \prec \in$.*

Axiom 5. *(Set Equality) It holds that $\forall U, V.(U = V) \Leftrightarrow (U \subset V) \wedge (V \subset U)$.*

Definition 6. *(Union) We define the symbol* $\cup$ *of type* $\mathrm{set} \times \mathrm{set} \rightarrow \mathrm{set}$. *Let A : set and B : set then we write $A \cup B$. It holds that $\forall U, V.(U \cup V) \Leftrightarrow (x \in U) \vee (x \in V)$. The precedence is $\cup \prec \subset$.*

Definition 7. *(Intersection) We define the symbol* $\cap$ *of type* $\mathrm{set} \times \mathrm{set} \rightarrow \mathrm{set}$. *Let A : set and B : set then we write $A \cap B$. It holds that $\forall U, V.(U \cap V) \Leftrightarrow (x \in U) \wedge (x \in V)$. The precedence is $\cap \prec \subset$. We group $\in, \subset, \cup$ and $\cap$ by set operators. The precedence is set operators $\prec$ logic operators.*

Figure 15.2: Example document in the text-editor TEXMACS

Step 1: Document Extraction

To extract the formal content of the document, we use a controlled authoring language to skip the burden of providing annotations, thus increasing the overall usability by dealing with pure TEXMACS documents. Let us note that the extraction process is not static, as the author usually introduces new notation during the theory development, which needs to be parsed at some point. Essentially, there are three options:

(i) The mediator parses the document completely and sends the information in a structured format to the proof assistant. The proof assistant in turn has only to check the incoming information.

(ii) The mediator parses only an outer syntax of the document, e.g., the theory structure, and sends the formulas to be parsed to the proof assistant.

(iii) The proof assistant parses the document completely and provides a structured version of the document for the mediator.

Each scenario has its advantages and disadvantages in terms of efficiency, dependencies being introduced, and implementation effort. For example, the proof assistant usually provides already parsers for formulas and algorithms to analyze them, such as typing algorithms. Moreover, these algorithms are not generic, but dependent on the logic the proof assistant is based upon. On the other hand, by relying on the language of the used proof assistant, it is difficult to support functionalities that are beyond its scope, such as a more flexible input language.

To completely abstract from the concrete form of the input language of the text-editor we decided in favor of (i). Let us note that the structure of documents in TEXMACS is considerably different from the document structure of ΩMEGA theory files and the corresponding parsers needed to be written anyway. Moreover, we wanted to be able to extend the document language independent of the language of the proof assistant, aiming at supporting natural language eventually.

It turned out in the beginning that because of the dynamic behavior of a mathematical theory a naive implementation of the initial parsing of the document using only a LALR parser was to inefficient[1]. Therefore, we employ more sophisticated parsing algorithms, such as combinator parsers and parsers that can dynamically be extended without being recompiled, such as directly-executable variants of an Earley parser [AH01] or operator precedence parsers. Essentially, the extraction process now works as follows: First of all, the document is preprocessed and split into segments almost corresponding to sentences. Then the following steps are incrementally performed for each segment: (1) the static parts of the authoring language, i.e., the controlled phrase structure, are parsed using a precompiled LALR(1) parser; (2) the dynamic parts of the authoring language, i.e., the formulas and notations, are parsed using a theory-specific Earley parser; (3) the segment is propagated to the proof assistance system. Note that the dynamic parts are always strictly separated from the static parts in the document because they are written inside a math mode macro.

For the example given above, the extracted theory is shown in Figure 15.3. For convenience, we present it already in a syntax suitable for the proof assistant, even though the internal representation of the mediator is slightly more complex. Note that notational information is not propagated, but completely maintained by the mediator, which sends already parsed formulas in an abstract syntax.

theory *Simple Sets*
newtype *elem*
newtype *set*
definition *element* $\in$*:elem*set->bool*
definition *subset* $\subset$*:set*set->bool* $\forall U, V. U \subset V \Leftrightarrow (\forall x. x \in U \Rightarrow x \in V)$
axiom *set equality* $\forall U, V. (U = V) \Leftrightarrow (U \subset V) \wedge (V \subset U)$
definition *union* $\cup$*:set*set->bool* $\forall U, V, x. x \in U \cup V \Leftrightarrow x \in U \vee x \in V$

definition *intersection* $\cap$*:set*set->bool* $\forall U, V, x. x \in U \cap V \Leftrightarrow x \in U \wedge x \in V$

Figure 15.3: Extracted document

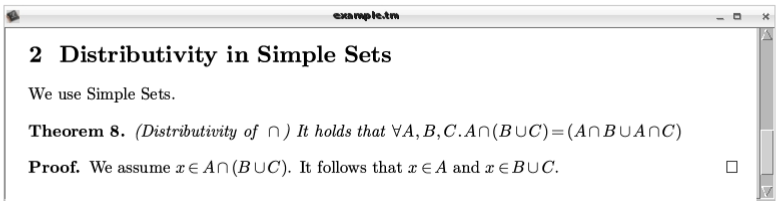
The theory is uploaded to the proof assistant, which constructs the specified theory. However, as the document does not contain any proof information, no verification process is started.

Step 2: Modification of the Document

Let us now continue our example by adding a new theory to the document, as shown in Figure 15.4(a). The new theory extends the previous theory by a theorem, for which a partial proof is given. Being uploaded, the mediator analyzes the parts of the document which have changed and recognizes the part that has been added. The new part is extracted (see Figure 15.4(b)), and the update sent to the reasoner.

Let us remark that while it is reasonable for the mediator to analyze the changes and to keep it as local as possible, it is the proof assistant that must actually perform

[1]In a first implementation, the processing of the above example took more than one minute.


example.tm

2 Distributivity in Simple Sets

We use Simple Sets.

Theorem 8. *(Distributivity of $\cap$) It holds that $\forall A,B,C.A\cap(B\cup C)=(A\cap B\cup A\cap C)$*

Proof. We assume $x\in A\cap(B\cup C)$. It follows that $x\in A$ and $x\in B\cup C$. □

(a) Added part of the document

theory *Distributivity in Simple Sets* **imports**
Simple Sets
theorem $\forall A,B,C.A\cap(B\cup C)=(A\cap B\cup A\cap C)$
proof
  assume $x\in A\cap(B\cup C)$
    $x\in A\wedge x\in(B\cup C)$ **by** . **from** .
  thus . **by** .
qed **by** .

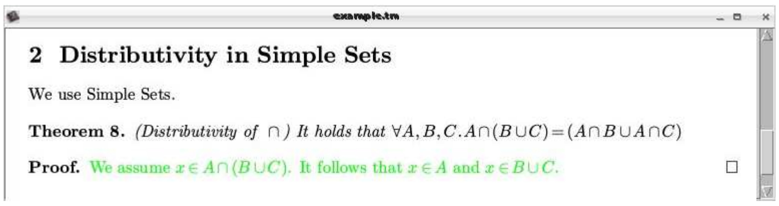
(b) Example proof in abstract syntax

Figure 15.4: Extended proof document and extracted part

the changes, as it is the only instance that can provide the final correctness guarantee. Moreover, while locality of changes is very desirable, management of change is currently not supported by most of the interactive theorem provers. Within our reasoner, we developed an interface to handle such differences. To keep the interface small, we limited the differences to the reasonable level of proof steps and formulas, such that deep changes in a formula are handled as a complete modification of the formula. The propagation of changes is essential for this real-time application because complete re-transformation and re-verification slows down the response time too much.

Step 3: Proof Checking and Status Updates

The reasoner updates its theory data structure, that is, starts a new theory and inserts the theorem together with its partial proof in the form of a proof plan. The verification is not started directly, but a corresponding expansion task maintained within the proof plan. The expansions are performed asynchronously. Whenever a step could be verified, a status update is sent to the mediator, which triggers the text-editor to mark the step as being proved, indicated by a green color. For our example, this results in the proof document shown in Figure 15.5.


example.tm

2 Distributivity in Simple Sets

We use Simple Sets.

Theorem 8. *(Distributivity of $\cap$) It holds that $\forall A,B,C.A\cap(B\cup C)=(A\cap B\cup A\cap C)$*

Proof. We assume $x\in A\cap(B\cup C)$. It follows that $x\in A$ and $x\in B\cup C$. □

Figure 15.5: Propagated Status Update inside the text-editor TEXMACS.

15.2.1 Proof Script Generation, Granularity Change, and Completion

Whenever a part of the proof is changed by the proof assistant, it must be propagated back to the mediator. In principle, the prover can construct arbitrary large parts and multiple hierarchies during the repair phase, which need to be inserted to the proof document when the user requests a more detailed variant of the proof. Proofs are always communicated in the form of declarative proof scripts, which can always be extracted, even in the case of procedural proofs (see Chapter 8). As default the most abstract proof hierarchy is communicated as a proof script to the mediator. However, the mediator can ask for a more detailed or a more abstract version of the proof script. Given a selected proof hierarchy, each proof step has to be transformed into a command of the proof script language.

15.2.2 Discussion

Usually, the author of a document writes many different proofs for different theorems in different theories before the document is finished. This corresponds to multiple, parallel proof attempts, which are maintained in ΩMEGA by the development graph, which is also responsible to determine the assertions that are available for one particular proof attempt. This functionality was not supported in the previous version of ΩMEGA. Having that infrastructure in place was the key to turn the ΩMEGA system into a server that can provide mathematical assistance services for multiple documents in parallel sessions.

The next step was to provide a facility to exchange proofs between the text-editor and the reasoner. To that end, we developed a declarative proof language and mechanisms to check the resulting proof obligations efficiently. A key step here was to map declarative proof commands to proof plan constructors, which model the verification of a proof step as proof refinement. Within proof plans, the execution of the refinement can be performed independently of the change of the proof plan, resulting in an asynchronous proof checker.

Having a basic proof checker available, an issue is that the author changes his document usually many times. Hence the proof assistance system has to be able to deal efficiently with non-monotonic changes not only inside a theory but also within the proofs. For instance, deleting an axiom from a theory should result in pruning or at least invalidating proof steps in all proofs that relied on that axiom. Furthermore, if by such an action a proof of some theorem is invalidated, then all other proof steps that used this theorem must be flagged and invalidated in turn. While the immediate "solution", i.e., to automatically re-execute all proof procedures, works in principle – at least if it is performed asynchronously – the local solution speeds up the rechecking and, as a consequence, an updated version of the document is obtained more quickly. Moreover, based on the relation between assertions and inferences, as well as the dependency information from the development graph, we could extend the management of change from theorems to the validation of individual proof steps.

Concerning the different representations of a proof document, let us point out that ΩMEGA stores a proof at different levels of granularity, which arise due to refinement operations. In contrast, a document is displayed at one particular granularity at the side of the text-editor. To consistently link the proof in the document with the respective part of the much more elaborate proof representation $\mathcal{PDS}$ in ΩMEGA the development of the notion of a "$\mathcal{PDS}$ view" (see Chapter 8) was essential.

To be able to change the granularity within the text-editor, it was necessary to insert new proof parts that were generated by ΩMEGA to the text-editor. Therefore, we developed mechanisms to convert procedural proofs to declarative proofs (see Chapter 8), resulting in a clean communication format in the form of declarative proofs. Of course, at some point, the resulting proofs may become too complex. A possible solution to this problem is to use granularity analysis techniques, as developed by Schiller [Sch10].

Finally, within the setting of interactive theorem proving, a user wants not only to specify proofs, but also proof strategies that extend the proof checker. To that end, we developed an independent strategy language (see Chapter 10). The language can be used within the proof document, and the conversion/compilation in the programming language of the theorem prover performed within the proof assistant. Moreover, declarative strategies (see Chapter 10.2) explicitly provide control over the granularity of the resulting proof scripts. As a result, we developed the first proof assistant that fully supports the document-centric approach to formalizing mathematics.

15.3 Related Work

MIZAR: The most prominent system for the publication of machine checked mathematics is MIZAR [TB85] with one of the largest libraries of formalized mathematics. As we have already compared the MIZAR language with our proof language in Section 8.5, we focus here on the overall workflow. MIZAR works in batch mode, that is, the user prepares an article, compiles it and loops both steps until there is no error reported. The MIZAR checker always processes the complete document, independent of errors it possibly encounters. One such processing cycle essentially corresponds to the initial upload of the proof document within our approach. In addition, we support changing the document without the need to recheck the complete document. This is particularly useful within our approach, as we support the automatic refinement of underspecified proofs, i.e., the verification of proofs that contain gaps, which are not supported by MIZAR. Moreover, our proof checker is extensible by the possibility to specify proof strategies within the proof document – a feature that is also not supported by MIZAR. Finally, we provide two facilities to automate the generation of declarative proof scripts, either by declarative tactics or by converting procedural proofs to their declarative counterpart, as described in Section 8.4.

PROOFGENERAL/ISABELLE: PROOFGENERAL [Asp00] is a generic front-end for proof assistants based on the text-editor Emacs. It provides a prover independent communication protocol between reasoners and displays and supports user interaction with a single focus of attention. Given an initial proof document, the document is uploaded to the reasoner, which parses the document and provides back a structured version of the document. Within the display, the unstructured version is then replaced by the structured version using the protocol functions `remove` and `newobj`. Proof steps are processed by moving the pointer forwards. Only unprocessed parts can be edited by the user. In such a case, PROOFGENERAL determines the parts that have been changed/inserted, and requests the reasoner to parse those parts in order to obtain a structured version of them. The checking of a proof step is asynchronously performed by the reasoner. However, it stops at the first error of the document, in which case the pointer of attention cannot be moved further forwards. PROOFGENERAL is mainly used in the context of ISABELLE, even though other proof assistants, such as PHOX, LEGO or COQ, are supported as well. In addition to verifying proof steps, ISABELLE uses the counterexample-generator NITPICK [BN10] to find counter models in the case that the conjecture is not provable.

Similar to our approach, ProofGeneral is based on a generic protocol to exchange information between a text-editor (display) and a reasoner. However, in the case of ProofGeneral, the input document must conform with the document format that is supported by the underlying reasoner. This needs not necessarily be the case in our approach, at the cost that our mediator must implement its own parsing algorithms which are not needed within ProofGeneral. Similar to our approach, ProofGeneral determines changes of the document locally. However, this functionality can only be used for parsing, but not for processing, which is always with respect to the current execution point. Even though the underlying protocol provides the possibility to insert parts within the proof script, this functionality has not yet been used by Isabelle's automated reasoning procedures or Isaplanner. It is one essential feature that is supported within our approach.

Matita: The Matita system "is meant to be first of all an interface between the user and the mathematical library" (see [ACTZ07] p. 3). As such, it stores proofs and defined concepts independently in a library, indexes it, and allows for efficient knowledge retrieval from the library. Every time a new mathematical concept is created and saved by the user it gets indexed, and becomes immediately visible in the library. To keep the library consistent, a mathematical concept and those depending on it are invalidated when the concept is changed or removed. The corresponding proofs need to be regenerated to verify if they are still valid. Matita thus allows the change of theorems and concepts locally, similar to our approach.

Sad: Sad [VLP07] is a system for the automated processing of formal mathematical texts written in a special language called ForTheL ({For}mal {The}ory {L}anguage) or in a traditional first-order language. Similar to our approach, the extraction of a formal representation of the document is separated from its verification. ForTheL is intended to be close to natural language. For example, the sentence "Let S be a set. A subset of S is a set X such that every element of X is in S" is a correct sentence in ForTheL and translated to $\forall S.\, \text{aSet}(S) \Rightarrow \forall x_1.\, \text{aSubsetOf}(x_1, S) \Leftrightarrow \text{aSet}(x_1) \wedge \forall x_2.\, \text{aElementOf}(x_2, x_1) \Rightarrow \text{aElementOf}(X_2, S)$ based on grammar rules [a subset/subsets of x][2] that are added at the beginning of the document. The result of the document extraction is a *normalized* version of the input document, which can be seen to be similar to our structured proof scripts. In a next step, the statements are reduced to goal statements and processed one by one by a reasoner component, thereby being simplified in various ways: "to decompose it to a number of smaller subtasks (e.g., to split a conjunctive goal or to perform a case analysis), to filter the set of premises, to generate suitable instantiations of relevant lemmas, to expand definitions, and so on" (see [LPV06] p. 563).

With respect to our approach, there are the following differences: (i) Even though our parsing techniques are general (see [Wag10]), in the current implementation the input language is not as close to natural language as the one of SAD, even though it could probably be expressed within the framework without much work. (ii) SAD runs in batch mode. Therefore, it does not provide any possibilities for reasoners to propagate back information to the initial proof document, nor does it provide any tools to support interactive proof construction. Rather, it is a proof checker that can handle large gaps. Moreover, as it runs in batch-mode, it always expects complete documents as input. (iii) The proving process is completely hidden in SAD. In particular, the user cannot control

[2]Equivalent forms of the same word are listed with slashes

the proof search process directly in the form of hints to restrict the available premises. This also implies that there is no mechanism to name formulas. Also, there is no possibility to extend the underlying reasoning procedures in form of user-defined tactics. (iv) SAD is based on untyped first order logic. In contrast, we use a simply typed higher-order logic.

NAPROCHE: The NAPROCHE project [KCKS09] ({Na}tural language {Pro}of {Che}cking) studies the semiformal language of mathematics (SFLM) as used in mathematical journals and textbooks with the goal of being able to extract their formal content into a first order logic and check them subsequently. A NAPROCHE text is structured by structure markers: Axiom, Theorem, Lemma, Proof, Qed and Case. These markers determine the macro structure of the document. The micro structure consists of statements and assertions, which combine terms with natural language, such as in "For all x, y, if $x \in y$ and $Ord(y)$ then $Ord(x)$.". The checking algorithm maintains a list of currently available facts which are updated during the checking. The checking itself is performed by invoking a classical first order reasoner. To keep the checking feasible, a premises selection algorithm tries to select only the relevant premises for an obligations. The NAPROCHE follows the approach of the SAD project and therefore differs from our work in the same way.

LOGIWEB: LOGIWEB [Gru07] is a system that supports publication and storage of machine checked mathematics, as well as scientific articles, and computer programs. We focus on the part that deals with storage of machine checked mathematics, which was originally designed to support map theory. Other logical foundations can be specified as axiomatic theories in the Hilbert style, i.e., by a (large) set of axiom schemes and a (small) set of inference rules. In addition to mathematical theories, documents can contain specification of tactics as well as proofs. A proof consists of a sequence of proof lines consisting of a label, an inference rule, and a formula that is justified by the rule. In that sense, LOGIWEB does not come along with a proof language in the traditional sense. Even though the system is extensible by user-written tactics, let us point out that there is no special tactic language. Rather, tactics are written in a general purpose programming language provided by LOGIWEB. An outstanding feature of LOGIWEB is its possibility to publish documents in a variety of formats, and access documents over the Internet. To that end, it strictly separates the input from the output document and provides several rendering facilities for an input document. Both input and output language can be extended. LOGIWEB works in batch mode, which differs from our work in the points mentioned above. To support a distributed library, articles that have been published cannot further be changed. Therefore, there is no need to consider the problem of rechecking parts, as considered by our approach.

15.4 Summary

In this chapter we discussed the integration of ΩMEGA into the scientific text-editor TEXMACS. While the developed architecture is mainly interesting from an implementational point of view, our experiments demonstrate that it is reasonable to communicate changes at the level of proof scripts. Moreover, our approach to translate proof scripts to proof plans has resulted in the new multi-focus paradigm where the user can change any part of the proof document.

Part V

Conclusion

16 Comparison with the previous ΩMEGA system

At an abstract level, the organization of the search of the automated reasoning system developed for this thesis is in the tradition of knowledge based proof planning [MS99b] as implemented by the previous ΩMEGA system [SBF+03] and its proof planner MULTI [MM05a]. However, the underlying logic, which was a variant of a higher-order natural deduction calculus, was exchanged for the CORE calculus. Moreover, many concepts have been enhanced in order to increase the usability and efficiency of the system. The goal of this chapter is to take a step back and discuss the most important changes and their impact.

16.1 Logical Foundations

The most obvious difference between the systems is the exchange of the underlying logic from a higher-order variant of natural deduction to the CORE calculus. This exchange required the adaptation of all reasoning procedures as well as the integration of external systems. There are two motivations for it:

(i) The proofs developed by a mathematician and the proofs developed by a student in a mathematical tutoring system are typically written at an argumentative level, i.e., as *proofs at the assertion level* with different types of *underspecification*. While so far such assertion level proofs needed to be constructed from the underlying ND-calculus proof, there was the educated guess that such proofs could be directly supported on top of the CORE calculus. Whether there are also practical benefits for automated reasoning procedures was not known.

(ii) The necessity that abstract proof plans need eventually to be expanded to calculus level proofs introduced certain constraints that had to be respected at the planning level, which are summarized in [BMM+01] as follows:

- Proof planning is goal centered and therefore backward reasoning is preferred. The proof construction in ND consist of an interplay of forward and backward steps, the order of steps is important for a successful proof. A specific order of proof steps during the proof search can lead to artificial problems with the eigenvariable condition and makes the management of hypotheses difficult. In particular, it might require backtracking.

- The requirements of the ND-calculus influence the design of methods and the heuristics to control the proof search. Hence, there is a danger that the motivation to capture the reasoning of mathematicians becomes secondary.

It was expected that the less restrictive calculus CORE would free the proof planning level from these constraints and that it is possible to define an independent planning layer on top of it.

Within the new system, the search for a proof is directly done at the assertion level on top of the CORE calculus. To that end, we introduced an assertion level interface that hides the internal representation of the CORE calculus and imposes further structure on the proof search. This has both computational benefits as well as representational benefits: Only the part of the proof state corresponding to the conjecture to be proved is shown, while the underlying background theory is hidden from the proof state and transformed to inferences. An inference application is mapped to a sequence of resolution replacement rules, and the multiplicities of the involved quantifiers are automatically adapted. Inferences are guaranteed to be correct and need not be expanded further. Therefore, no logical constraints corresponding to the expansion of methods need to be considered when defining strategic reasoning procedures on top of the inferences. Further control information can conveniently be attached locally to inferences to further structure the search space. Moreover, we could show the following:

- It is possible to translate assertion level proofs to declarative proof scripts, which are extensively used within the interactive theorem proving community. Our experiments confirm Huang's insight that the assertion level provides a suitable basis for proof presentation. It is important to point out that this feature is not supported by state of the art interactive theorem provers, such as ISABELLE.

- Using the assertion level as the logical basis for automated reasoning procedures is beneficial for specific mathematical domains. Our experiments in Section 12 show that a simple assertion level prover outperforms state of the art theorem provers in the domain of set theory and can solve several problems that are out of the scope of many automated theorem provers. This was not expected beforehand, in particular because the used techniques are general purpose and rather domain independent. Moreover, we could show that the assertion level is also useful for the verification/reconstruction of proof steps in the context of proof tutoring. The abstract nature of the assertion level allows the formulation of a simple depth bounded breadth-first search algorithm to verify the student steps efficiently. Moreover, ambiguities as well as incomplete proof steps can be reconstructed, which is probably not possible without further ado when relying on pure logical reasoning. The resulting proof reconstructions obtained by the algorithm provide the basis for further analysis of the student utterance, as studied by Schiller [Sch10] in the context of the DIALOG project.

Thus, there are several reasons why proof search at the assertion level is beneficial. However, was the change of the calculus inevitable? The answer is no, at least for assertion level proof search without deep inference: In [AD09] we have shown that it is also possible to directly search at the assertion level within the framework of natural deduction or sequent calculi by lifting assertions to the level of inference rules. Similar to the CORE interface, an assertion application corresponds to a sequence of reasoning steps in the underlying calculus. The problem of permutability of proof steps (see [BMM+01]) in the presence of meta-variables can be solved using a more liberalized δ-rule to reduce the dependencies introduced by the Eigenvariable condition, as shown in the paper.

The insight that the assertion level can be supported by different calculi is indeed interesting and it identifies the assertion level to be a rather calculus independent layer. Nevertheless, the use of CORE as underlying reasoning framework was beneficial because:

- CORE supports contextual reasoning on subformulas. We have shown in Chapter 13 that this allows the construction of exponentially shorter proofs compared to the sequent calculus without cut. Moreover, the possibility to reason on subformulas had an extensive impact on the design of the strategy language, which now provides first class support for subformula reasoning.

- Our theoretical and practical investigations increased the understanding of resolution replacement rules and its relation to simplification methods as used in tableau systems.

- The uniform notation used by CORE provides an abstraction from the actual formula language and characterizes the relationship of subformulas based on their behavior rather than based on their syntactic structure. As a result, proofs become invariant under certain equivalence transformations, as discussed in Section 3.1.

- On the implementational level, the change of the underlying calculus has led to an intermediate interface that makes the planning layer independent of the calculus.

However, we have also to admit that the design of an assertion level interface on top of the CORE calculus is a non-trivial task. This is because the unstructured representation of the proof state as a single formula can result in large structures, in particular in the presence of large theories. This has the following drawbacks:

- Large structures are rather unsuited for communication with a user.

- The generality of replacement rules results in a high non-determinism which needs to be controlled within the automated reasoning procedures. Without further restrictions, the high redundancy leads to efficiency problems within the proof search algorithms. Moreover, the application of replacement rules can become to complex to be understood by a human. As inferences are based on these replacement rules, this property is inherited by inferences.

- Transformations are always global with respect to the complete formula and difficult to analyze.

It is the structuring mechanism on top of the calculus which allows for a local analysis of the effects of an assertion application and for a translation of generated proofs to declarative proof scripts. The locality properties lead to an efficient computation of the effects of resolution replacement rules and allow the study of redundancy criterions and heuristically motivated search space restrictions at the level of inference rules. Therefore, reasoning procedures can cover the full range from deep inference to shallow inference and be adapted to the needs of the underlying application.

16.2 Knowledge Representation and Maintenance

In this section, we discuss several shortcomings of the previous ΩMEGA system with respect to knowledge management, before highlighting the main enhancements to overcome these problems.

16.2.1 Different Form of Knowledge

In ΩMEGA, several different forms of the same knowledge existed: In addition to the mathematical theory, the system provided *tactics* for interactive proof construction, *methods*, *control rules* and *strategies* for proof planning, and *agents* for resource-bounded reasoning. While the knowledge representation techniques were general purpose from an abstract point of view, their concrete specification needed to be given in the underlying programming language of the proof system, which was Lisp. This approach has several drawbacks:

Usability: The necessity to manually encode the mathematical knowledge in different formats to be able to use the underlying system in its full functionality puts a substantial burden on the user, which prevents its use by non-experts. The fact that the knowledge needs to be encoded in the underlying programming language of the system makes the situation even worse. An indicator of this burden can be taken from the limit domain, which consisted of more than 17000 lines of manually written code for control rules, strategies, methods, and agents. Moreover, as the knowledge was stored in a format that was difficult to read, many proof methods were redundant, and it was difficult for a user to find the method that was appropriate for a specific situation.

Evaluation: The programming language makes it sometimes difficult to judge the success of proof planners, as it makes it difficult to easily perceive the knowledge encoded in the proof planning operators: "The devil is in the detail, that is, it is always possible to hide the crucial creative step (represented as a specific method or represented in the object language by an appropriate lemma) and to pretend a level of generality that has not actually been achieved. To evaluate a solution, all tactics, methods, theorems, lemmata and definitions have to be made explicit." (see [SBA06] p. 549).

Maintenance: Explicit different forms of the mathematical knowledge complicate its maintenance and its development.

Flexibility vs. Efficiency: The separation of knowledge in methods, control rules, and strategies allows the flexible design and combination of different algorithms. While the generality is remarkable, it also introduces efficiency problems when being implemented without further optimization: In general, there are many different types of choice points within the proof search, and each choice point gives rise to a new type of control rule. Every time such a choice point is reached, all available control rules need to be evaluated. As control rules were interpreted in the previous ΩMEGA system, this process was rather slow.

Semantics of Control: There is no semantics that specifies the interplay of several control rules.

16.2.2 Methods: Original Idea and Practice

The original idea of knowledge based proof planning was to reason with complex proof operators that imitate human problem solving behavior: "The representation of a proof, at least while it is developed, consists of a sequence of complex operators, such as the application of a homomorphism, the expansion of a definition, the application of a lemma, some simplification, or the differentiation of a function. Each of these operators, called methods, can in principle be expanded into a sequence of inference steps, say, of a natural deduction (ND) calculus by a tactic." (see [MS99b] p. 70). The following classification of the existing methods shows that this goal could only partially be reached:

Methods as techniques: Complex-Estimate is a typical example for a method that represents a domain specific mathematical reasoning technique for the domain of real numbers. The goal of the method is to estimate the magnitude of the absolute value of complex terms by representing it as a linear combination of an existing assumption, which was computed by a CAS. To guarantee the correctness of this method, it needed to be expanded. Methods like ComplexEstimate meet the original motivation.

Methods as assertions: Many methods correspond to a special application of a theorem: A classical example is a method that initiates a proof by induction, or methods like Solve* that reduce a goal $t_1 < t_2$ with the help of an assumption $t'_1 < t'_2$, to a subgoal $t'_2\sigma < t_2\sigma$, provided that t_1 and t'_1 are unifiable by a substitution σ. Because this method corresponds to the transitivity of $<$, the method can be guaranteed to be correct, once transitivity of $<$ has been shown. The additional information that is contained in the method, but not within the theorem in its general form, is the requirement that the conclusion as well as one premise needs to be instantiated. While the correctness differs from above, it can also be seen to be a complex proof operator, at least when compared with calculus rules.

Methods to encode logical low-level reasoning: Using natural deduction as the underlying logical formalism, it was not possible to support the abstract reasoning process without considering logical details. Therefore, there are methods that reflect calculus level inference rules. As such, these methods were guaranteed to be correct. Examples are methods for $\forall_I, \exists_I, \wedge_I$. While being necessary, these methods are not in the spirit of the original motivation for proof planning.

Methods to encode control flow: Methods are also used to encode the control flow that cannot be expressed otherwise. An example is the method SetFocus-B that highlights a subformula in an assumption to be extracted in a subsequent step, as well as the TellCS to pass a constraint to the constraint solver. As these methods do not contribute to the proof from a mathematical point of view, it is questionable whether these operations meet the original motivation for a method.

16.2.3 Limitations of Strategic Reasoning in Multi

Cooperation of strategies: Strategies are not allowed to call other strategies in the proof planner Multi, s. Therefore, cooperation between two strategies s_1 and s_2 was only possible by stopping the strategy s_1 during its execution, invoking s_2, and finally reinvoking s_1 after successful termination of s_2. This mechanism was error-prone as it required complicated data structures.

An example (see [Mei03] p. 154) is the interplay between the strategies SolveInequality, NormalizeLineTask, and UnwrapHyp for the limit domain. SolveInequality is a strategy that is applicable to formulas which are inequalities or can be reduced to inequalities and comprise knowledge of how to solve such them. UnwrapHyp is a strategy that uses various other methods such as $\wedge_E$ and $\Rightarrow_E$ to extract the subformula in the assumption that has been highlighted using the method SetFocus-B. NormalizeLineTask is a strategy that decomposes a goal involving quantifiers and connectives using methods reflecting calculus rules such as $\forall_I, \exists_I, \wedge_I$.

On an abstract level, the cooperation works as follows: For complex goals that contain inequalities as subformulas SolveInequality interrupts and places a demand for the strategy NormalizeLineTask on the control blackboard. As a consequence, MULTI selects the specified strategy within the next step to decompose the complex goal, before reinvoking the original strategy. Switching from SolveInequality to UnwrapHyp is triggered when the goal cannot be solved by available assumptions. In this case, the method SetFocus-B selects an inequality in an assumption. Subsequently, a control rule signals SolveInequality to interrupt and to invoke UnwrapHyp.

Such a control flow has several drawbacks:

(i) Interruption and reinvokation of strategies is difficult to visualize. Indeed, this might have been a reason why strategy applications were not represented in the $\mathcal{PDS}$. However, this information is needed for the generation of abstract declarative proof scripts, as shown in Section 8.4.

(ii) From a technical point of view, interruption and reinvokation of a strategy requires additional data structures to implement such a functionality, thus increasing the complexity of the underlying implementation.

(iii) In MULTI, the choice of which strategy to apply next is not predefined, but determined by *strategic meta-reasoning*: After all applicable strategies have been determined and corresponding job offers placed on the control blackboard, strategic control rules are evaluated on the job-offers to determine the strategy to be applied next. There is always only a single meta-reasoner and meta-reasoning is always global.

Adaptability of strategies: In MULTI, the methods and control rules that are used by a strategy are fixed. This rigidity prevents strategies to adapt themselves to the global context in which they are invoked. For example, after proving a specific theorem, simplification should take the theorem into account, provided it is of the desired structure.

Limitations of Proof Plans: In the previous ΩMEGA system it is not possible to explore or present two proof ideas in parallel.

16.2.4 Improved Knowledge Representation and Maintenance: Document-Centric Proof Planning

From a conceptual point of view, there are the following main enhancements of proof planning in MULTI:

Full support of the document-centric approach: The central medium within which proof search is organized is a proof document. It contains all relevant knowledge, such as constants and types declarations, theorems including their proofs, and even proof search procedures in the form of strategies. This has been achieved by the introduction of a strategy language as an intermediate layer to bridge the gap between the reasoning procedures and the programming language. This has the following advantages:

- The reasoning becomes rather independent of a particular implementation, and by using the assertion level as a basis for the proof search even independent of the underlying logic.
- The possibility to declaratively specify conditions on proof states in the form of various patterns allows the user to characterize the proof situations in which a strategy should be applied in a compact form. By relying on compilation techniques, this allows for a generic specification of strategies and control information as before, but additionally results in optimized Lisp code that can be executed efficiently. In particular, heuristic decisions that are local to an inference are also evaluated locally.
- Having the document as single knowledge source makes the used knowledge explicit and therefore allows for a better evaluation of the proof planning approach.

New Methodology: Within the new version of ΩMEGA, the basic proof operators are inferences, which are always guaranteed to be correct and automatically synthesized from domain assertions. As a consequence, there is no need to encode such methods manually, which drastically increases the usability of the system[1]. Inferences can be annotated with further control information to restrict their applicability and are automatically synthesized from assertions. Compared to MULTI, annotated inferences thus correspond to proof planning methods for assertions. There are also inferences dealing with calculus steps, such as the introduction of a case split. However, explicit decomposition of formulas at the planning level is not needed due to the possibility to reason on subformulas. Moreover, as no quantifiers are present, they do not have to be eliminated. The requirement of methods to be correct avoids techniques such as ComplexEstimate to be formulated as inference and introduces a clear separation between strategies and inferences. Methods to encode control flow are abandoned; they are no longer necessary due to an improved control flow.

Optimized Control Flow: Combination of different strategies is possible in a more powerful way: strategies can invoke strategies and therefore cooperate in a hierarchical fashion. As a consequence, no interruption/reinvokation of strategies is necessary for a combination of different strategies. Moreover, within the same hierarchy, strategies can be combined using the combinators of the strategy language or declarative proof scripts. Therefore, it is possible to speak about the future of derivation, which cannot easily be modeled with control rules. A simple example is an induction followed by a simplification, or the complex strategy factorbound which we have designed in the context of the limit domain.

[1]Even though not all knowledge encoded in the proof planner MULTI has been transferred to the new system, a formalization that can handle the LIM+ problem and contains the factorbound methods consists only of several hundred lines compared to the 17000 lines before

First class support of declarative proof scripts: We provide first class support for declarative proof scripts: Proof scripts can interactively be developed within the proof document, as for example in Mizar or Isabelle. In addition, there are two possibilities to obtain such scripts from automated procedures: (i) Assertion level proofs as well as procedural proofs can be translated to declarative proof scripts. (ii) Declarative proof scripts are directly constructed by declarative strategies, which not only construct such proofs, but are also specified in a declarative way.

Besides the key changes as stated above, there are the following two minor changes:

Support Of Proof Alternatives The representation of alternative proof attempts within one proof object is supported by the new $\mathcal{PDS}$. This functionality is extensively used in the context of proof tutoring to represent ambiguities that could not be resolved during the proof search, as described in Chapter 11. Moreover, it extends the system by the possibility to support different search modes, such as e.g. breadth-first search, which becomes useful when searching at the abstract assertion level and optimality guarantees (with respect to the proof length) are needed, such as in the context of proof tutoring.

Explicit Knowledge Filtering To support such knowledge filtering explicitly, our new strategy language supports the filtering of knowledge as an explicit step. As a consequence, it is possible to define adaptive strategies, in particular, it also allows a strategy to take advantage of the structure of the theory, such as to use only assertions defined within the current theory and extend them if the proof cannot be found.

16.3 Integration of External Reasoners

Knowledge based proof planning has been shown to provide a natural framework for the integration of external systems (see e.g. [KKS98]) . Because of its importance, we subsequently discuss what kind of external systems were linked in the previous Ωmega system, and whether the integration has been taken over to the new system.

Integration of Constraint Solvers

The previous Ωmega system is connected to the constraint solver Cosie [ZM04], which is used to construct mathematical objects with theory-specific properties as witnesses for free (existential) variables, as well as to detect inconsistencies of constraints to prune the search. Cosie is able to solve inequalities and equations over the field of real numbers by applying constraint simplification rules in the form of conditional rewriting rules. The main reason for developing an individual constraint solver was the desire to support user-defined functions, such as "min", in answer constraints.

As in the previous Ωmega system, the new system postpones the instantiation of universally quantified variables until many or all conditions of the constraints are available or until the constraints determine an instantiation. However, we either rely on already existing implementations, such as the quantifier elimination methods for the reals implemented in Qepcad [Bro03], Reduce or Mathematica, or specify simplification rules within the proof document to support user defined functions. Indeed, all simplification rules used by Cosie (see the appendix of [ZM04]) can easily be specified within the proof document. This has the advantage that the constraint simplification is extensible during the development of the theory, and that the constraint solving behavior can be made explicit as a proof strategy within the document.

Integration of Computer Algebra Systems

The previous ΩMEGA system has been connected to computer algebra systems (CAS) via the SAPPER module (see [KKS98]). For such a connection, the main issue that arises in practice is how to provide a correctness guarantee of a result that has been obtained by the CAS. While in principle several CAS could easily be connected to SAPPER, it turned out that a "standard CAS could not be used for the integration, since such a system provides answers, but no justifications [..]" (see [KKS98] p. 10-11). Consequently, a hand-crafted CAS μCAS was developed that provided sufficiently rich explanation traces. While this approach showed the feasibility of such an integration and the reconstruction of proofs from traces, the requirement to implement a CAS manually overrides the advantage that existing operational knowledge and systems can be integrated.

Within the new system, we have integrated the computer algebra system MAXIMA and REDUCE. The new approach is not to rely on traces, but on general strategies to verify the answer. In the domain of reals, a factorization provided by the CAS is verified by a decision procedure that has been encoded in the underlying strategy framework. The main advantages are: (i) there is a guarantee that a solution can be checked, provided it was correct; (ii) existing systems can be integrated without reimplementing the underlying algorithms. Note that this approach is feasible in situations where checking a solution is simpler than finding it, such as the equality between a factorization and its original form.

Another possibility in the spirit of μCAS would be to specify and verify the underlying algorithms of a CAS once and for all within the theorem prover and using a reflection principle.

Integration of Automated Theorem Provers

Within MULTI, tasks can be sent off to the resolution based automated theorem provers OTTER, BLIKSEM, SPASS, PROTEIN, and to the equational provers EQP and WALDMEISTER, and the resulting proof objects are translated back intro assertion level ND-proofs. These are then integrated back into the $\mathcal{PDS}$ using the mediator TRAMP [Mei00a], provided that no meta-variables were used. Up to now, none of these systems has been integrated within the new architecture. The reason for this is that the TRAMP system, which is needed to perform the transformation of the proof object to an assertion level proof is no longer maintained by its developer. However, conceptually, there is in principal no problem for such an integration, as the assertion level is directly supported by the new ΩMEGA. Even more, the integration should become much simpler, as assertion level proofs are first class objects within the new architecture and need not to be expanded to ensure their correctness.

17 Conclusion and Future Work

In this thesis, we presented assertion level proof planning with compiled strategies. Our objective was to make the mechanization of mathematics more natural by being able to verify human-style proofs, as well as by providing tools to automatically construct abstract proofs similar to those in a textbook. Thereby, we focused on a document-centric approach, in which the proof document builds the central medium where tools which assist the author are made available. We approached this goal from two directions:

- We lifted the underlying calculus to a more abstract level by installing the assertion level as a theoretical basis for the proof search and combined it with the deep inference paradigm. The motivation was the expectation that by reducing the gap between formal calculus proofs and informal textbook proofs, verification of informal proofs as well as presentation of automatically generated calculus-level proofs would become easier. Moreover, for proof tutoring, we had to face the result of Schiller (see [Sch05]) that the number of calculus steps of a derivation in natural deduction do not provide a reliable measure for the granularity of a proof step, i.e., the argumentative size of a proof step, which is an important measure for proof granularity (see [Sch05]). It was expected that a more abstract calculus would be more suitable for the automation of this task.

  The results of this thesis are: It is possible to search for a proof directly at the assertion level and to combine it with the deep inference feature. This results in shorter proofs with smaller proof states which are therefore more readable. The proofs can directly be translated to declarative proof scripts, which is only useful if the underlying calculus is at an appropriate granularity. Moreover, our experiments in the domain of set theory indicate that automated reasoning procedures benefit from the abstract search at the assertion level. The implemented assertion level prover outperforms state of the art theorem provers by speed and success. Similarly, our results considering the class of Statman tautologies indicate that automated reasoners can take advantage of the deep inference mechanism. In the domain of proof tutoring, the abstract nature of the assertion level provides a possibility to reconstruct ambiguous and incomplete student proof steps by a simple depth bounded forward search that cannot be automated by pure logical means. Moreover, based on our work, Schiller has shown (see [Sch10] p. 146) that in the domain of set theory a single assertion level step is judged by experts to be of appropriate step size in ninety percent of the cases.

- We examined whether it is possible to separate the tactic language from the programming language by extending proof documents to also contain specifications of tactics. This is indeed possible: We designed an intermediate tactic language which allows the declarative specification of procedural as well as declarative strategies. Its main features are: (i) A query language to *dynamically retrieve* relevant knowledge from structured theories, resulting in *adaptive proof strategies*; (ii) Language constructs to control the deep application of inferences; (iii) Horizontal as well as vertical *refinements of proof sketches* being expressed in a declarative language; (iv) *Dynamic declarative proof scripts* as declarative specification language of proof plans; (v) *Efficiency* by *compiling* the control information to executable programs. In particular, our language provides new foundational ideas to automate the declarative style of proof. This represents a major step to overcome the main deficiency of declarative proof scripts, namely that they are laborious to write. Our case studies in the context of proof tutoring and the limit domain indicate the superiority of the language over generic programming languages, as common patterns of reasoning can be specified within the proof document, the specification costs are significantly reduced measured in lines of code, and the resulting tactics are highly efficient. This not only increases the usability of interactive theorem provers and proof planners, but also provides a possibility to evaluate systems that are based on heuristics, as all knowledge becomes explicit. Moreover, the separation of the strategy language from the programming language brings benefits for the maintenance of systems, as optimizations can easily be incorporated and the compiler can treat local decisions locally. We strongly believe that our language provides a vehicle to teach common patterns of reasoning, and that intermediate tactic languages open up a new perspective towards exchanging reasoning procedures between different proof assistants in the long-term view.

17.1 Future Work

The achievements of this thesis point to several interesting directions for future work in different areas: proof checking, proof automation, proof strategies, proof tutoring, and fundamental research.

Proof Checking: We have seen that proof planning provides a suitable framework for the implementation of proof checkers, as underspecified declarative proof scripts can easily be translated to proof plans. Proof search can then be used to obtain a machine-checkable proof by refinement. It would be interesting to incorporate further proof strategies that support the automatic refinement of proof sketches, such as specific decision procedures, as well as computer algebra systems. This also includes the integration of already existing theorem provers by reimplementing the TRAMP system for the new calculus. Two interesting case studies to evaluate these strategies would be to reconsider a formalization of Landau's book [Lan30], as in the AUTOMATH project, or to consider automatic refinements of the proof sketches given in [Wie05].

Another promising aspect to speed up the processing within an interactive setting is the integration of management of change at the level of proof scripts, as done by Schairer (see

[Sch06]) at the level of proof trees. This seems particularly plausible taking the relation between proof scripts and proof trees into account, as well as the close correspondence between assertions and inferences. Further speed-ups could be achieved by parallelizing the checking process.

Proof Automation: Our results indicate that proof search at the assertion level is superior in areas where machine oriented provers suffer from a "lack of focus". A natural question is to what extent these techniques carry over to other domains. While the developed techniques are of a general nature, further case studies are needed to investigate the effectiveness of our techniques in other domains. Moreover, as the framework was intended to be used within the interactive setting rather than to provide an efficient stand-alone theorem prover, there are several opportunities to further improve the performance of the prover. These include the use of term indexing techniques instead of traversal functions for the matching process, the incorporation of techniques to avoid backtracking, as well as the change of the underlying programming language.

Proof Strategies: We have shown that it is possible to design a strategy language that is independent of the programming language of the proof assistant. This suggests to provide compilation functions for existing proof assistants, such as Isabelle or HOL Light. Aside from making a powerful strategy language available in systems that do not provide an abstract strategy language, this would be a step towards the long-term goal of sharing tactics/proof strategies between different proof assistants. Another next step is to extend the underlying ideas to the specification language layer and to allow the specification of tactics that generate or transform specifications, but also tactics that transform both specifications and proof scripts in combination. Finally, it is attractive to extend the notion of proof plans to heterogeneous proof plans, i.e., proof plans that comprise subtasks in different logics and can make use of logic translations to solve a given problem efficiently. An ideal environment to investigate these ideas is provided by the Hets system (see for example [Mos02]), which provides an abstract framework for the specification of logics and logic translations.

Proof Tutoring: We have shown that it is possible to reconstruct human proof steps efficiently in the context of proof tutoring in the domain of set theory. In addition to widening the application domain to other areas, such as Analysis, further work is necessary to automate strategic teaching decisions based on our automatic reconstruction, as well as to evaluate the system in a real classroom setting. This also includes the modeling of typical failures, e.g., by buggy rules, which can easily be integrated within our framework. An interesting task would be to try to recognize the strategy a student is following by using techniques developed in the context of plan recognition and to lift the notion of a buggy rule to the strategic level, i.e., to introduce buggy strategies.

Fundamental Research: Our results indicate the usefulness of the combination of assertion level reasoning and deep inference. While the transfer of the assertion level to the sequent calculus or natural deduction is immediate, the deep inference application requires further work. The main step would be to simulate the application of a resolution replacement rule in the sequent calculus, e.g., with a cut.Moreover, it is worthwhile to study further proof theoretic properties within the sequent calculus, such as cut elimination in the context of derived rules, as well as to provide a constructive method to transform a given derivation in the sequent calculus to an assertion level proof, making use of permutation properties. Further investigations are also necessary to study completeness of the calculus in the higher-order setting, e.g., by considering results from higher-order tableaux. Another direction should be to minimize the logical kernel of the system.

Index

Bibliography

[ABB93] J. R. Anderson, F. S. Bellezza, and C. F. Boyle. The geometry tutor and skill acquisition. In J. R. Anderson, editor, *Rules of the Mind*, chapter 8. Erlbaum, 1993.

[ABB00] P. B. Andrews, M. Bishop, and C. E. Brown. System Description: TPS: A Theorem Proving System for Type Theory. In D. McAllester, editor, *Automated Deduction, CADE-17 (CADE-00) : 17th International conference on Automated Deduction ; Pittsburgh, PA, USA, June 17-20, 2000*, volume 1831 of *Lecture notes in computer science*, pages 164–169. Springer, 2000.

[ABD$^+$06] S. Autexier, C. Benzmüller, D. Dietrich, A. Meier, and C.-P. Wirth. A generic modular data structure for proof attempts alternating on ideas and granularity. In *Proc. of MKM*, pages 126–142, Bremen, 2006. Springer.

[ABDS08] S. Autexier, C. Benzmüller, D. Dietrich, and J. Siekmann. Resource adaptive processes in automated reasoning systems. In Matthew Crocker and J. Siekmann, editors, *Resource Adaptive Cognitive Processes*, LNAI, pages 389–423. Springer, 2008.

[ABDW08] S. Autexier, C. Benzmüller, D. Dietrich, and M. Wagner. Organisation, transformation, and propagation of mathematical knowledge in omega. *Mathematics in Computer Science*, pages 253–277, 2008. DOI 10.1007/s11786-008-0054-6.

[ABEG95] S. Abdennadher, F. Bry, N. Eisinger, and T. Geisler. The theorem prover satchmo : strategies, heuristics and applications. In Jean-Jacques Chabrier, editor, *JFPLC*, pages 349–355, 1995.

[ABFL06] S. Autexier, C. Benzmüller, A. Fiedler, and H. Lesourd. Integrating proof assistants as reasoning and verification tools into a scientific wysiwig editor. In David Aspinall and C. Lüth, editors, *Proc. of UITP'05*, ENTCS, pages 16–39, january 2006.

[ABI$^+$96] 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*, 16:321–353, 1996.

[ABP$^+$04] P. B. Andrews, C. E. Brown, F. Pfenning, M. Bishop, S. Issar, and H. Xi. Etps: A system to help students write formal proofs. *Journal of Automated Reasoning*, 32:75–92, 2004.

[ACP01] A. Abel, B. E. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. In Uwe Egly, A. Fiedler, Helmut

Horacek, and Stephan Schmitt, editors, *Proc. of the Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, pages 33–48. Universitá degli studi di Siena, June 2001.

[ACTZ07] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[AD06] S. Autexier and D. Dietrich. Synthesizing proof planning methods and oants agents from mathematical knowledge. In Jon Borwein and Bill Farmer, editors, *Proc. of MKM'06*, volume 4108 of *LNAI*, pages 94–109. Springer, august 2006.

[AD09] S. Autexier and D. Dietrich. Atomic metadeduction. In Bärbel Mertsching, editor, *Proc. 32nd Annual German Conference on Artificial Intelligence. German Conference on Artificial Intelligence (KI-09), September 15-18, Paderborn, Germany*, pages 444–451. Springer, 9 2009.

[AD10a] S. Autexier and D. Dietrich. Recent developments in omega's proof search programming language. In *Programming Languages for Mechanized Mathematics Systems*, pages 52–59, 2010.

[AD10b] S. Autexier and D. Dietrich. A tactic language for declarative proofs. In Kaufmann and Paulson [KP10], pages 99–114.

[ADG$^+$01] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer-Verlag.

[ADL10] D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proofs. *Mathematics in Computer Science*, 3:309–330, M.h 2010.

[ADW10] S. Autexier, D. Dietrich, and M. Wagner. Proof search formalisms and grammar formalisms in omega. In *Workshop on Mathematically Intelligent Proof Search*. 2010.

[AF06] S. Autexier and A. Fiedler. Textbook proofs meet formal logic - the problem of underspecification and granularity. In M. Kohlhase, editor, *Proc. of MKM'05*, volume 3863 of *LNAI*, IUB Bremen, Germany, january 2006. Springer.

[AGC$^+$04] A. Asperti, F. Guidi, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. A content based mathematical search engine: whelp. In *In: Post-proceedings of the Types 2004 International Conference, Vol. 3839 of LNCS*, pages 17–32. Springer-Verlag, 2004.

[AH77a] K. Appel and W. Haken. Every planar map is four-colorable, ii: Reducibility. *Illinois J. Math*, 21:491–567, 1977.

[AH77b] K. Appel and W. Haken. The solution of the four-color map problem. *Sci. Amer.*, 237:108–121, 1977.

[AH97] M. Archer and C. Heitmeyer. Human-style theorem proving using PVS. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275, pages 33–48, Murray Hill, NJ, 1997. Springer-Verlag.

[AH01] J. Aycock and R. N. Horspool. Directly-executable earley parsing. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 229–243, London, UK, 2001. Springer-Verlag.

[AHMS02] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In Hélène Kirchner and C. e Ringeissen, editors, *Proc. 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*, pages 495–501. Springer, September 2002.

[ALW06] D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In M. Kohlhase, editor, *Mathematical Knowledge Management MKM 2005*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 65–80. Springer, 2006.

[ALW07] D. Aspinall, C. Lüth, and D. Winterstein. A framework for interactive proof. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2007.

[And72] P. B. Andrews. General models, descriptions, and choice in type theory. *The Journal of Symbolic Logic*, 37:385–397, 1972.

[And80] P. B. Andrews. Transforming matings into natural deduction proofs. In *CADE*, pages 375–393, 1980.

[And81] P. B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[And91] P. B. Andrews. More on the problem of finding a mapping between clause representation and natural deduction representation. *J. Autom. Reasoning*, 7(2):285–286, 1991.

[And92] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[AP92] F. Andersen and K. D. Petersen. Recursive boolean functions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proc. of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, pages 367–377. IEEE Computer Society Press, 1992.

[APCS01] A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. Helm and the semantic math-web. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2001.

[ARC09] A. Asperti, W. Ricciotti, and C. Sacerdoti Coen. A new type for tactics. In *ACM SIGSAM PLMMS*, pages 22–29, 2009.

[ASDP90] P. B. Andrews, I. Sunil, N. Dan, and F. Pfenning. The TPS Theorem Proving System. In Stickel [Sti90], pages 641–642.

[Asp00] D. Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[Aug85] L. Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[Aut03] S. Autexier. *Hierarchical Contextual Reasoning.* PhD thesis, Computer Science Department, Saarland University, Saarbrücken, Germany, 2003.

[BB77] A. M. Ballantyne and W. W. Bledsoe. Automatic proofs of theorems in analysis using nonstandard techniques. *J. ACM*, 24(3):353–374, 1977.

[BB78] J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155–191, 1978.

[BBH72] W. W. Bledsoe, R. S. Boyer, and W. H. Henneman. Computer proofs of limit theorems. *Artif. Intell.*, 3(1-3):27–60, 1972.

[BBHI03] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning.* Cambridge University Press, 2003.

[BBK04] C. Benzmüller, C. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.

[BCZ98] A. Bauer, E. M. Clarke, and X. Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[BDH$^+$99] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors. *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*. Springer, 1999.

[BDP89] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In A. H. Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 1–30. Academic Press, New York, 1989.

[BDSA07] C. Benzmüller, D. Dietrich, M. Schiller, and S. Autexier. Deep inference for automated proof tutoring? In J. Hertzberg, M. Beetz, and R. Englert, editors, *KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI, KI 2007, Osnabrück, Germany, September 10-13, 2007, Proceedings*, pages 435–439. Springer, 2007.

[BDW07] P. Brauner, G. Dowek, and B. Wack. Normalization in supernatural deduction and in deduction modulo. available at http://hal.inria.fr/ inria-00141720, 2007.

[Bec97] B. Beckert. Semantic tableaux with equality. *J. Log. Comput.*, 7(1):39–58, 1997.

[Bec03] B. Beckert. Depth-first proof search without backtracking for free-variable clausal tableaux. *J. Symb. Comput.*, 36(1-2):117–138, 2003.

[Bee92] M. Beeson. Mathpert: Computer support for learning algebra, trig, and calculus. In A. Voronkov, editor, *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 454–456. Springer, 1992.

[Bee98] M. Beeson. Automatic generation of epsilon-delta proofs of continuity. In Calmet and Plaza [CP98], pages 67–83.

[BEF00] P. Baumgartner, N. Eisinger, and U. Furbach. A confluent connection calculus. In Steffen Hölldobler, editor, *Intellectics and Computational Logic*, volume 19 of *Applied Logic Series*, pages 3–26. Kluwer, 2000.

[Ben99] C. Benzmüller. *Equality and Extensionality in Higher-Order Theorem Proving.* PhD thesis, Department of Computer Science, Saarland University, 1999.

[Ber37] P. Bernays. A system of axiomatic set-theory. *Journal of Symbolic Logic*, 2:65–77, 1937.

[Bet65] E. W. Beth. *The foundations of mathematics: a study in the philosophy of science.* Studies in logic and the foundations of mathematics. North-holland, 2nd rev. ed. edition, 1965.

[BFG$^+$03] C. Benzmüller, A. Fiedler, M. Gabsdil, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, D. Tsovaltzi, B. Quoc Vo, and M. Wolska. Tutorial dialogs on mathematical proofs. In *Proc. of IJCAI-03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, pages 12–22, Acapulco, Mexico, 2003.

[BFN96] P. Baumgartner, U. Furbach, and I. Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orlowska, editors, *JELIA*, volume 1126 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[BG07] P. Bruscoli and A. Guglielmi. On the proof complexity of deep inference. In *Proof, Computation, Complexity (PCC)*, pages 1–32, 2007.

[BGLS92] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In Deepak Kapur, editor, *Proc. of the CADE-11*, volume 607 of *LNAI*, pages 172–192, Saratoga Springs, NY, June 1992. Springer.

[BGML01] L. Bachmair, H. Ganzinger, Second Readers D. Mcallester, and C. Lynch. Resolution theorem proving, 2001.

[BGvW96] R. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:9–467, 1996.

[BH80] W. W. Bledsoe and L. M. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In W. Bibel and R. A. Kowalski, editors, *CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 1980.

[BH98] B. Benhamou and L. Henocque. Finite model search for equational theories (fmset). In Calmet and Plaza [CP98], pages 84–93.

[BHC95] C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between isabelle and maple. In *ISSAC*, pages 150–157, 1995.

[BHK07a] P. Brauner, C. Houtmann, and C. Kirchner. Principles of superdeduction. In *LICS*, pages 41–50. IEEE Computer Society, 2007.

[BHK07b] P. Brauner, C. Houtmann, and C. Kirchner. Superdeduction at work. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 132–166. Springer, 2007.

[BHKK$^+$07a] C. Benzmüller, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, and M. Wolska. Natural Language Dialog with a Tutor System for Mathematical Proofs. In Ruqian Lu, Jörg Siekmann, and Carsten Ullrich, editors, *Cognitive Systems*, volume 4429 of *LNAI*. Springer, 2007.

[BHKK$^+$07b] C. Benzmüller, H. Horacek, I. Kruijff-Korbayová, M. Pinkal, J. Siekmann, and M. Wolska. Natural language dialog with a tutor system for mathematical proofs. In *Proc. of the 2005 joint Chinese-German conference on Cognitive systems*, pages 1–14, Berlin, Heidelberg, 2007. Springer-Verlag.

[BHL$^+$06] C. Benzmüller, H. Horacek, H. Lesourd, I. Kruijff-Korbayova, M. Schiller, and M. Wolska. A corpus of tutorial dialogs on theorem proving; the influence of the presentation of the study-material. In *Proc. of International Conference on Language Resources and Evaluation (LREC 2006)*, Genova, Italy, 2006. ELDA.

[BHS93] B. Beckert, R. Hähnle, and P. H. Schmitt. The even more liberalized δ-rule in free variable semantic tableaux. In G. Gottlob, A. Leitsch, and D. Mundici, editors, *Proceedings, 3rd Kurt Gödel Colloquium (KGC), Brno, Czech Republic*, LNCS 713, pages 108–119. Springer, August 1993.

[Bib81] W. Bibel. On matrices with connections. *Journal of ACM*, 28:633–645, 1981.

[Bib87] W. Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 2 edition, 1987.

[Bil96] J.-P. Billon. The disconnection method - a confluent integration of unification in the analytic framework. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *TABLEAUX*, volume 1071 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 1996.

[Bil05] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[BJD98] B. Buchberger, T. Jebelean, and D.Vasaru. Theorema: A System for Formal Scientific Training in Natural Language Presentation. In , editor, *Proc. of Ed-Media 1998 (International Conference on Educational Multimedia)*, pages 174–179, Freiburg, Germany, June 20-23 1998.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. An Overview of the Theorema Project. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC'97)*, pages 384–391, Hawaii, 1997. ACM Press.

[BK97] P. Borovanský and H. Kirchner. Strategies of elan: meta-interpretation and partial evaluation. In *Proc. of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97, Workshops in Computing*, Amsterdam, The Netherlands, September 1997. Springer-Verlag.

[BK98] C. Benzmüller and M. Kohlhase. LEO – a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *Proc. of the 15th International Conference on Automated Deduction (CADE-15)*, number 1421 in LNAI, pages 139–143, Lindau, Germany, 1998. Springer.

[BK07] G. Burel and C. Kirchner. Cut elimination in deduction modulo by abstract completion. In Sergei N. Artemov and Anil Nerode, editors, *LFCS*, volume 4514 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2007.

[BKKR01] P. Borovansky, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[BKS06] D. Bryce, S. Kambhampati, and D. E. Smith. Planning graph heuristics for belief space search. *J. Artif. Intell. Res. (JAIR)*, 26:35–99, 2006.

[BKT94] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer Berlin / Heidelberg, 1994.

[BL94] M. Baaz and A. Leitsch. On skolemization and proof complexity. *Fundam. Inform.*, 20(4):353–379, 1994.

[Ble86] W. W. Bledsoe. The use of analogy in automatic proof discovery. Technical report, Microelectronics and Computer Technology Corporation, 1986.

[Ble90] W. W. Bledsoe. Challenge problems in elementary calculus. *J. Autom. Reasoning*, 6(3):341–359, 1990.

[BM72] R. S. Boyer and J. S. Moore. The sharing of structure in theorem–proving programs. In *Machine Intelligence*, chapter 7, pages 101–116. J. Wiley and Sons, New York, 1972.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[BM06] E. Balland and P.-E. Moreau. Optimizing pattern matching by program transformation. *Electronic Communications of the EASST*, 2006.

[BMM$^+$01] C. Benzmüller, A. Meier, E. Melis, M. Pollet, and V. Sorge. Proof planning: A fresh start? In *Proc. of the IJCAR 2001 Workshop: Future Directions in Automated Reasoning*, pages 25–37, Siena, Italy, 2001.

[BN98] F. Baader and T. Nipkow. *Term* Rewriting *and* All That. Cambridge University Press, 1998.

[BN10] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Kaufmann and Paulson [KP10], pages 131–146.

[Boo47] G. Boole. *The Mathematical Analysis of Logic, Being an Essay Toward a Calculus of Deductive Reasoning.* Macmillan, Cambridge, 1847.

[Boo58] G. Boole. *The Laws Of Thought.* Dover Publications,Inc., New York, dover edition, 1958. originally published by Macmillan in 1854.

[Bou07] E. Bouwers. Improving automated feedback-building: A generic rule-feedback generator. Master's thesis, Utrecht University, 2007.

[BP95] B. Beckert and J. Posegga. leantap: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995.

[BR03] G. Bancerek and P. Rudnicki. Information retrieval in mml. In *Proc. of MKM'03*, pages 119–132, London, UK, 2003. Springer-Verlag.

[Bro03] C. W. Brown. Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM BULLETIN*, 37:97–108, 2003.

[Bro06] C. E. Brown. Verifying and invalidating textbook proofs using scunak. In Jonathan M. Borwein and William M. Farmer, editors, *MKM*, volume 4108 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 2006.

[Bro11] C. Brown. Satallax. http://www.ps.uni-saarland.de/ cebrown/satallax/index.php, 2011.

[Bru73] N. G. De Bruijn. AUTOMATH - Ein Projekt zur Kontrolle von Mathematik. In P. Braffort, editor, *Proc. of the symposium APLASM*, volume I, Orsay, France, 1973. Talk given at Innsbrucker Mathematikertag, 1974. German translation of "The AUTOMATH Mathematics Checking Project".

[BS82] R. Bartle and D. Sherbert. *Introduction to Real Analysis.* Wiley, 1982.

[BS94] F. Baader and J. Siekmann. Unification theory. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming*, pages 41–125. Oxford University Press, Inc., New York, NY, USA, 1994.

[BS01a] F. Baader and W. Snyder. Unification theory. In Robinson and Voronkov [RV01], pages 445–532.

[BS01b] C. Benzmüller and V. Sorge. Ω-Ants – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Proc. of Calculemus-2000*, pages 81–97, St. Andrews, UK, 2001. AK Peters.

[BTPF08] C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, P. Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[Buc01] B. Buchberger. The pcs prover in theorema. In *Computer Aided Systems Theory - EUROCAST 2001-Revised Papers*, pages 469–478, London, UK, 2001. Springer-Verlag.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

[Bun96] A. Bundy. Rippling: Greatest hits. Research paper, University of Edinburgh, 1996.

[Bun98] A. Bundy. A science of reasoning. *Lecture Notes in Computer Science*, 1397, 1998.

[Bun99] A. Bundy. A survey of automated deduction. In *Artificial Intelligence Today*, pages 153–174. 1999.

[Bun02] A. Bundy. A critique of proof planning. In *Computational Logic: Logic Programming and Beyond*, pages 160–177. Springer, 2002.

[BV80] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. In *Cognitive Science*, volume 4, pages 379–426, 1980.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In Stickel [Sti90], pages 647–648.

[BvHHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1991.

[BW05] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philosophical transactions - Royal Society. Mathematical, physical and engineering sciences*, 363(1835):2351–2375, 2005.

[CAB$^+$86] R. L. Constable, S. F. Allen, H. M. Bromley, W. 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 Development System.* Prentice-Hall, NJ, 1986.

[Cai05] P. Cairns. Alcor: A user interface for mizar. *Mechanized Mathematics and its Applications*, 4:83–88, 2005.

[Car84] L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217, 1984.

[CBSB96] F. Cantu, A. Bundy, A. Smaill, and D. A. Basin. Experiments in automating hardware verification using inductive proof planning. In *Formal Methods in Computer-Aided Design, First International Conference*, volume 1166 of LNCS, pages 94–108. FMCAD96, 1996.

[CC99a] O. Caprotti and A. M. Cohen. Connecting proof checkers and computer algebra using openmath. In Bertot et al. [BDH$^+$99], pages 109–112.

[CC99b] O. Caprotti and A. M. Cohen. Integrating computational and deduction systems using openmath. *Electr. Notes Theor. Comput. Sci.*, 23(3):469–480, 1999.

[CDF$^+$07] C. B. Callaway, M. Dzikovska, E. Farrow, M. Marques-Pita, C. Matheson, and J. D. Moore. The beetle and beediff tutoring systems. In *Proc. of the 2007 Workshop on Spoken Language Technology for Education (SLaTE)*, Farmington, Pennsylvania, USA, September 2007.

[CfW04] L. Cruz-filipe and F. Wiedijk. A decision procedure for equational reasoning in commutative algebraic structures, 2004.

[CG07] P. A. Cairns and J. Gow. Integrating searching and authoring in mizar. *J. Autom. Reasoning*, 39(2):141–160, 2007.

[Chr93] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10:95–113, 1993.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 1936.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[CKKS10] M. Cramer, P. Koepke, D. Kühlwein, and B. Schröder. Premise selection in the naproche system. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 434–440. Springer, 2010.

[Coq03] Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA, 1999-2003. See `http://coq.inria.fr/doc/main.html`.

[Cor94] Intel Corporation. Statistical analysis of floating point flaw. White Paper, 1994.

[Cor07] P. Corbineau. A declarative language for the coq proof assistant. In Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors, *TYPES*, volume 4941 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2007.

[CP98] J. Calmet and J. A. Plaza, editors. *Artificial Intelligence and Symbolic Computation, International Conference AISC'98, Plattsburgh, New York, USA, September 16-18, 1998, Proceedings*, volume 1476 of *Lecture Notes in Computer Science.* Springer, 1998.

[CS00] L. Cheikhrouhou and V. Sorge. PDS – a three-dimensional data structure for proof plans. In *Proc. of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA 2000)*, pages 143–148, march 2000.

[CS03] K. Claessen and N. Sörensson. New techniques that improve mace-style finite model finding. In *Proc. of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, pages 427–442, 2003.

[Da97] B. I. Dahn and al. Integration of automated and interactive theorem proving in ilf. In *Proc. of CADE-14*, pages 57–60, 1997.

[Dav81] M. Davis. Obvious logical inferences. In *Proc. of the 7th IJCAI*, pages 530–531, 1981.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61, 1970.

[dB73] N. G. de Bruijn. AUTOMATH, A Language for Mathematics. Séminaire de Mathématiques Superieures 52, Département de Mathématiques, Université de Montréal, Montréal, Canada, 1973.

[dB94] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In R. P Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 865 – 935. Elsevier, 1994.

[DB07] D. Dietrich and M. Buckley. Verification of proof steps for tutoring mathematical proofs. In Rosemary Luckin, Kenneth R. Koedinger, and Jim E. Greer, editors, *AIED*, volume 158 of *Frontiers in Artificial Intelligence and Applications*, pages 560–562. IOS Press, 2007.

[DB09] D. Dietrich and M. Buckley. Verification of human-level proof steps in mathematics education. *Teaching Mathematics and Computer Science*, 2009. In print.

[DE98] M. D'Agostino and U. Endriss. WinKE: A Proof Assistant for Teaching Logic. In *Proc. of the First International Workshop on Labelled Deduction*, 1998.

[Del99] D. Delahaye. Information retrieval in a coq proof library using type isomorphisms. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 1956 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 1999.

[Del02] D. Delahaye. A Proof Dedicated Meta-Language. In *Proc. of Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark)*, volume 70 (2) of *ENTCS*. Elsevier, July 2002.

[Den04a] L. Dennis. What is the difference between a method and a tactic? Blue Book Note 1356, Edinburgh Dream Group, 2004.

[Den04b] L. Dennis. What is the difference between a method and a tactic ii, straying into what is proof planning and why do we do it? Blue Book Note 1381, Edinburgh Dream Group, 2004.

[DF06] L. Dixon and J. D. Fleuriot. A proof-centric approach to mathematical assistants. *J. Applied Logic*, 4(4):505–532, 2006.

[DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998.

[DHK03] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *J. Autom. Reasoning*, 31(1):33–72, 2003.

[Die06] D. Dietrich. The task-layer of the ΩMEGA system. Diploma thesis, FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2006.

[Dix05] L. Dixon. *A Proof Planning Framework for Isabelle.* PhD thesis, University of Edinburgh, 2005.

[DJ07] L. Dixon and M. Johansson. Isaplanner 2: A proof planner for isabelle, 2007.

[DJP06] L. A. Dennis, M. Jamnik, and M. Pollet. On the comparison of proof planning systems: lambdaclam, omega and isaplanner. *Electr. Notes Theor. Comput. Sci.*, 151(1):93–110, 2006.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DLM99] A. Degtyarev, A. Lyaletski, and M. Morokhovets. Evidence algorithm and sequent logical inference search. In *LPAR '99: Proc. of the 6th International Conference on Logic Programming and Automated Reasoning*, pages 44–61, London, UK, 1999. Springer-Verlag.

[DLS00] J. D. Richardson D. Lacey and A. Smaill. Logic program synthesis in a higher order domain. *Computational Logic*, 1861:87–100, 2000.

[DM94] M. D'Agostino and M. Mondadori. The taming of the cut. classical refutations with analytic cut. *J. Log. Comput.*, 4(3):285–319, 1994.

[DM05] D. Delahaye and M. Mayero. Dealing with algebraic expressions over a field in coq using maple. *J. Symb. Comput.*, 39(5):569–592, 2005.

[dN99] H. de Nivelle. Bliksem 1.10 user manual. Technical report, Max-Planck-Institut für Informatik, 1999.

[Dow09] G. Dowek. From proof theory to theories theory, 2009.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In Robinson and Voronkov [RV01], pages 535–610.

[DPT06] E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electronic Notes in Theoretical Computer Science*, 155:341 – 359, 2006. Proc. of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[DS08] D. Dietrich and E. Schulz. Crstal: A declarative language for the encoding of proof techniques. In *Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 16–28, 2008.

[DS09] D. Dietrich and E. Schulz. Integrating structured queries into a tactic language. *JAL - Special issue on Programming Languages and Mechanized Mathematics Systems*, pages 1–32, 2009.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[DSW08] D. Dietrich, E. Schulz, and M. Wagner. Authoring verified documents by interactive proof construction and verification in text-editors. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, Masakazu Suzuki, and F. Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2008.

[DT95] B. Drabble and A. Tate. O-plan mixed initiative planning capabilities and protocols. Technical report, University of Edinburgh, 1995.

[Dup00] D. Vasaru Dupre. *Automated Theorem Proving by Integrating Proving, Solving and Computing.* PhD thesis, RISC, Johannes Kepler University of Linz, 2000.

[FAR84] R. G. Farrell, J. R. Anderson, and B. J. Reiser. An interactive computer-based tutor for lisp. In *AAAI*, pages 106–109, 1984.

[FF95] E. E. Feigenbaum and Julian Feldman, editors. *Computers and Thought.* AAAI Press / The MIT Press, 1995.

[FGT92] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Imps: System description. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer, 1992.

[FGT93a] W. Farmer, J. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11, 1993.

[FGT93b] W. Farmer, J. Guttman, and F. J. Thayer. The imps user's manual. Technical Report M-93B138, The MITRE Corporation, November 1993.

[Fie01] A. Fiedler. *P.rex*: An interactive proof explainer. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning: Proc. of IJCAR'01*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer.

[Fil06] J.-C. Filliâtre. Backtracking iterators. In *Proc. of the 2006 workshop on ML*, ML '06, pages 55–62, New York, NY, USA, 2006. ACM.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving / 2nd Edition.* Springer-Verlag New York Inc., 1996. ISBN 0-387-94593-8.

[Fit97] M. Fitting. leantap revisited, 1997.

[FM88] A. Felty and D. Miller. Proof explanation and revision. Technical report, University of Pennsylvania, 1988.

[FM01] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *ICFP*, pages 26–37, 2001.

[Fra22] A. A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237, 1922.

[Fre79] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelprache des reinen Denkens.* Verlag Nebert, Halle, 1879.

[Gab00] D. M. Gabbay. *Goal-Directed Proof Theory.* Kluwer Academic Publishers, 2000.

[Gal97] D. Galmiche, editor. *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '97, Pont-à-Mousson, France, May 13-16, 1997, Proceedings*, volume 1227 of *Lecture Notes in Computer Science.* Springer, 1997.

[Gas09] H. Gast. Towards a modular extensible isabelle interface. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science.* Springer, 2009.

[GBC98] F. Giunchiglia, P. Bertoli, and A. Coglio. The omrs project: State of the art, 1998.

[GBK08] S. Gruttmann, D. Böhm, and H. Kuchen. E-assessment of mathematical proofs: Chances and challenges for students and tutors. In *Proc. of the 2008 International Conference on Computer Science and Software Engineering, Wuhan, China, December 12-14, 2008*, volume 5, pages 612–615. IEEE Computer Society, 2008.

[Geh94] W. Gehrke. Detailed catalogue of canonical term rewriting systems generated automatically, 1994.

[Gel59] H. Gelernter. Realization of a geometry-theorem proving machine. In *Proc. of an International Conference on Information Processing*, pages 273–282, Paris, France, 1959. reprinted in [SW83] and in [FF95].

[Gen69] G. Gentzen. *The Collected Papers of Gerhard Gentzen (1934-1938).* Edited by Szabo, M. E., North Holland, Amsterdam, 1969.

[Geu09] H. Geuvers. Proof assistants: history, ideas and future. *Sadahana, Special Issue on Interactive Theorem Proving and Proof Checking*, 34:3–25, 2009.

[GG04] A. Guglielmi and A. Guglielmi. Polynomial size deep-inference proofs instead of exponential size shallow-inference proofs, 2004.

[Gie01] M. Giese. Incremental closure of free variable tableaux. In *IJCAR '01: Proc. of the First International Joint Conference on Automated Reasoning*, pages 545–560, London, UK, 2001. Springer-Verlag.

[Gie02] M. Giese. *Proof Search without Backtracking for Free Variable Tableaux.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe, July 2002.

[Gie06] M. Giese. Saturation up to redundancy for tableau and sequent calculi. In Miki Hermann and Andrei Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2006.

[Gil60] P. C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM J. Res. Dev.*, 4(1):28–35, 1960.

[GL97] J. Grundy and T. Långbacka. Recording hol proofs in a structured browsable format. In Michael Johnson, editor, *AMAST*, volume 1349 of *Lecture Notes in Computer Science*, pages 567–571. Springer, 1997.

[GM06] H. Geuvers and L. Elie Mamane. A document-oriented coq plugin for texmacs, 2006.

[GMW79] M. J. Gordon, A. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer Verlag Germany, 1979.

[Gol73] A. Goldberg. *Computer assisted instruction: The application of theorem proving to adaptive response analysis.* PhD thesis, Stanford, 1973.

[Gol81] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

[Gor00] M. Gordon. From lcf to hol: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–186. The MIT Press, 2000.

[Grä91] A. Gräf. Left-to-right tree pattern matching. In Ronald V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 323–334. Springer, 1991.

[Gru07] K. Grue. The Layers of Logiweb. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, LNAI. Springer, June 2007.

[GS05] H. Ganzinger and J. Stuber. Superposition with equivalence reasoning and delayed clause normal form transformation. *Inf. Comput.*, 199(1-2):3–23, 2005.

[GUM$^+$04] G. Goguadze, C. Ullrich, E. Melis, J. Siekmann, C. Gross, and R. Morales. Leactivemath structure and metadata model. Technical report, Saarland University, 2004.

[Häh01] R. Hähnle. Tableaux and related methods. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press, 2001.

[Hal06] T. Hales. The kepler conjecture (eight years later)., 2006.

[Har96a] J. Harrison. Hol light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 265–269. Springer-Verlag, 1996.

[Har96b] J. Harrison. A mizar mode for hol. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 1996.

[Har96c] J. Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 1996.

[Har98] J. Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.

[Har06] J. Harrison. Formal verification in industry, 2006.

[Har09] J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[HC96] K. Homann and J. Calmet. Structures for symbolic mathematical reasoning and computation. In J. Calmet and Carla Limongelli, editors, *DISCO*, volume 1128 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 1996.

[Hen50] L. Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15:81–91, 1950.

[Her30] J. Herbrand. Recherches sur la theorie de la demonstration. *The Journal of Symbolic Logic*, 15:81–91, 1930.

[HF96] X. Huang and A. Fiedler. Presenting machine-found proofs. In *Proc. CADE-13, LNAI 1104*, pages 221–225. Springer Verlag, 1996.

[HHH07] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS*, pages 176–183. AAAI, 2007.

[Hin94] L. M. Hines. Str+ve and integers. In *CADE-12: Proc. of the 12th International Conference on Automated Deduction*, pages 416–430, London, UK, 1994. Springer-Verlag.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System description: Waldmeister — improvements in performance and ease of use. In H. Ganzinger, editor, *Proc. of the 16th Conference on Automated Deduction*, number 1632 in LNAI, pages 232–236. Springer, 1999.

[HK00] N. Heffernan and K. Koedinger. Building a 3rd generation ITS for symbolization: Adding a Tutorial Model with Multiple Tutorial Strategies. In *Proc. of the ITS Workshop in Algebra Learning*, 2000.

[HK02] N. T. Heffernan and K. R. Koedinger. An intelligent tutoring system incorporating a model of an experienced human tutor. In Stefano A. Cerri, Guy Gouardères, and Fábio Paraguaçu, editors, *Intelligent Tutoring Systems : 6th International Conference, ITS 2002, Biarritz, France and San Sebastian, Spain, June 2-7, 2002. Proceedings.*, pages 149 – 157. Springer, 2002.

[HKC98] X. Huang, M. Kerber, and L. Cheikhrouhou. Adaptation of declaratively represented methods in proof planning. *Annals of Mathematics and Artificial Intelligence*, 23(3–4):299–320, 1998.

[HMAE96] G. D. Hume, J. A. Michael, R. A. Allen, and M. W. Evens. Hinting as a tactic in one-on-one tutoring. *Journal of the Learning Sciences*, 5(1):23–47, 1996.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[Hof01] J. Hoffmann. Ff: The fast-forward planning system. *AI Magazine*, 22(3):57–62, 2001.

[Hom97] K. Homann. *Symbolisches Lösen mathematischer Probleme durch Kooperation algorithmischer und logischer Systeme*, volume 152 of *DISKI*. Infix Verlag, St. Augustin, Germany, 1997.

[HT93a] J. Harrison and L. Théry. Extending the hol theorem prover with a computer algebra system to reason about the reals. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *HUG*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184. Springer, 1993.

[HT93b] J. Harrison and L. Théry. Reasoning about the reals: The marriage of hol and maple. In A. Voronkov, editor, *LPAR*, volume 698 of *Lecture Notes in Computer Science*, pages 351–353. Springer, 1993.

[HT07] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, M.h 2007.

[Hua94a] X. Huang. Proverb - a system explaining machine-found proofs. In *Proc. of 16th Annual Conference of the Cognitive Science Society*, pages 427–432, 1994.

[Hua94b] X. Huang. Reconstructing proofs at the assertion level. In Alan Bundy, editor, *Proc. 12th CADE*, pages 738–752. Springer-Verlag, 1994.

[Hua96] X. Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[Hua99] X. Huang. The presentation of proofs at the assertion level, 1999.

[Hue76] G. Huet. *Résolution d'équations dans des langages d'ordre 1,2,...,ω*. PhD thesis, Université Paris VII, 1976.

[Hue97] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[HW05] H. Horacek and M. Wolska. Interpretation of Mixed Language Input in a Mathematics Tutoring System. In *Proc. of AIED-05 Workshop on Mixed Language Explanations in Learning Environments*, pages 27–34, 2005.

[Ire92] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *Logic Programming and Automated Reasoning*, pages 178–189, 1992.

[Ire96] A. Ireland. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[JC08] J. Kiniry J. Charles. A lightweight theorem prover interface for eclipse. In *User Interfaces for Theorem Proving*, 2008.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

[JKK$^+$05] W. A. Hunt Jr., M. Kaufmann, R. B. Krug, J. S. Moore, and E. Whitman Smith. Meta reasoning in acl2. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2005.

[KA93] K. Koedinger and J. R. Anderson. Reifying implicit planning in geometry: Guidelines for model-based intelligent tutoring system design. In S. P. Lajoie and S. J. Derry, editors, *Computers as Cognitive Tools*, pages 15–46. Routledge, 1993.

[KAB$^+$04] E. Klieme, H. Avenarius, W. Blum, P. Döbrich, H. Gruber, M. Prenzel, K. Reiss, K. Riquarts, J. Rost, H. Tenorth, and H. J. Vollmer. The development of national educational standards - an expertise. Technical report, Bundesministerium für Bildung und Forschung / German Federal Ministry of Education and Research, 2004.

[KAE$^+$10] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

[Kah08] O. Kahramanogullar. Maude as a platform for designing and implementing deep inference systems. *Electronic Notes in Theoretical Computer Science*, 219:35 – 50, 2008. Proc. of the Eighth International Workshop on Rule Based Programming (RULE 2007).

[Kal01] J. A. Kalman. *Automated Reasoning with OTTER*. Rinton Press, Paramus, NJ, 2001.

[KB70] D.E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KCKS09] D. Kühlwein, M. Cramer, P. Koepke, and B. Schröder. The naproche system. In *Intelligent Computer Mathematics*. Springer, 2009.

[Kel83] J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. In Raoul N. Smith, Richard W. Pew, and Ann Janda, editors, *CHI '83: Proc. of the SIGCHI conference on Human Factors in Computing Systems, Boston, Massachusetts, United States, December 12-15, 1983*, pages 193–196, New York, NY, USA, 1983. ACM.

[Ker98] M. Kerber. Proof planning: A practical approach to mechanized reasoning in mathematics. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, pages 77–95, 1998.

[KJ01] B. Konev and T. Jebelean. Solution Lifting Method for Handling Meta-variables in Theorema. In S. Maruster, B. Buchberger, V. Negru, and T. Jebelean, editors, *Proc. of SYNASC01*, pages 15–23, Timisoara, Romania, October 2001. Mirton.

[KKS98] M. Kerber, M. Kohlhase, and V. Sorge. Integrating computer algebra into proof planning. *J. Autom. Reasoning*, 21(3):327–355, 1998.

[Kle98] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *ESA*, volume 1461 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 1998.

[KMR05] O. Kahramanogullari, P.-E. Moreau, and A. Reilles. Implementing deep inference in tom. In *In Structures and Deduction (SD 05), ICALP Workshop*, 2005.

[KMW04] F. Kamareddine, M. Maarek, and J. B. Wells. Mathlang: Experience-driven development of a new mathematical language. *Electr. Notes Theor. Comput. Sci.*, 93:138–160, 2004.

[KN04] F. Kamareddine and R. Nederpelt. A refinement of de bruijn's formal language of mathematics. *Journal of Logic, Language and Information*, 13(3):287–340, 2004.

[KO99] C. Kreitz and J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[KP10] M. Kaufmann and L. C. Paulson, editors. *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010.

[KWHvR07] C. Kaliszyk, F. Wiedijk, M. Hendriks, and F. van Raamsdonk. Teaching logic usign a state-of-the-art proof assistant. In H. Geuvers and P. Courtieu, editors, *Proc. of the International Workshop on Proof Assistants and Types in Education*, pages 33–48, 2007.

[Lam93] L. Lamport. How to write a proof, 1993.

[Lan30] E. Landau. *Grundlagen der Analysis.* Chelsea Publishing Company, 1930.

[Ler09] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Let93] A. A. Letichevsky. Development of rewriting strategies. In *PLILP '93: Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 378–390, London, UK, 1993. Springer-Verlag.

[Lin89] C. Lingenfelder. Structuring computer generated proofs. In *IJCAI*, pages 378–383, 1989.

[LLB02] S. Lukins, A. Levicki, and J. Burg. A tutorial program for propositional logic with human/computer interactive learning. In Judith L. Gersting, Henry MacKay Walker, and Scott Grissom, editors, *SIGCSE*, pages 381–385. ACM, 2002.

[LMG94] R. Letz, K. Mayr, and C. Goller. Cotrolled integration of the cut rule into connection tableaux calculi. *J. Autom. Reasoning*, 13(3):297–337, 1994.

[LPV06] A. V. Lyaletski, A. Paskevich, and K. Verchinine. Sad as a mathematical assistant - how should we go from here to there? *J. Applied Logic*, 4(4):560–591, 2006.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. Setheo: A high-performance theorem prover. *J. Autom. Reasoning*, 8(2):183–212, 1992.

[LV97] S. P. Luttik and E. Visser. Specification of rewriting strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing*. Springer-Verlag, 1997.

[Mas98] F. Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In Harrie C. M. de Swart, editor, *TABLEAUX*, volume 1397 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1998.

[MBD03] S. Kambhampati M. B. Do. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[McC62] J. McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory*, volume 5 of *Proc. of Symposia in Pure Mathematics*, pages 219–227. AMS, Providence, Rhode Island, 1962.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

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

[McC03] W. McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[McD83] D. D. McDonald. Natural language generation as a computational problem. In Brady and Berwick, editors, *Computational Models of Discourse*. MIT Press, 1983.

[McM91] A. F. McMichael. Mechanization of analytic reasoning about sets. In *AAAI*, pages 427–433, 1991.

[Mei00a] A. Meier. System description: Tramp - transformation of machine-found proofs into natural deduction proofs at the assertion level. In *Proc. of the 17th International Conference on Automated Deduction, number 1831 in Lecture Notes in Artificial Intelligence*, pages 460–464. Springer-Verlag, 2000.

[Mei00b] A. Meier. Transformation of machine-found proofs into assertion level proofs. Technical report, Saarland University, 2000.

[Mei03] A. Meier. *Proof Planning with Multiple Strategies.* Phd thesis, FR 6.2 Informatik, Saarland University, 2003.

[Mei04] A. Meier. *Proof planning with multiple strategies.* PhD thesis, University of Saarland, FB Informatik, 2004.

[Mel98a] E. Melis. Ai-techniques in proof planning. In *ECAI*, pages 494–498, 1998.

[Mel98b] E. Melis. The "limit" domain. In *AIPS*, pages 199–207, 1998.

[Mel05] E. Melis. Design of erroneous examples for activemath. In Ch.-K. Looi and G. McCalla, editors, *Proc. of the 12th International Conference on Artificial Intelligence in Education (AIED 2005).*, volume 125, pages 451–458. IOS Press, 2005.

[MFS02] E. Maclean, J. Fleuriot, and A. Smaill. Proof planning non-standard analysis. *7th International Symposium on AI and Mathematics*, 2002.

[Mil84] D. Miller. Expansion tree proofs and their conversion to natural deduction proofs. In Robert E. Shostak, editor, *CADE*, volume 170 of *Lecture Notes in Computer Science*, pages 375–393. Springer, 1984.

[MK98] P.-E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 230–249. Springer, 1998.

[MLK96] J Strother Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division algorithm. *IEEE Transactions on Computers*, 47, 1996.

[MM05a] A. Meier and E. Melis. Failure reasoning in multiple-strategy proof planning. *Electr. Notes Theor. Comput. Sci.*, 125(2):67–90, 2005.

[MM05b] A. Meier and E. Melis. System description: Multi a multi-strategy proof planner. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[MMS08] E. Melis, A. Meier, and J. H. Siekmann. Proof planning with multiple strategies. *Artif. Intell.*, 172(6-7):656–684, 2008.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L-Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.

[MN90] U. Martin and T. Nipkow. Ordered rewriting and confluence. In Mark E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 1990.

[MOMV05] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, M.h 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

[Mor47] A. De Morgan. *Formal Logic.* Taylor and Walton, 1847.

[Mos02] T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In *Proc. of FOSSACS'02*, LNCS, pages 326–341. Springer, April 2002.

[MP09] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[MPS02] A. Meier, M. Pollet, and V. Sorge. Comparing approaches to the exploration of the domain of residue classes. *Journal of Symbolic Computation Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34:287–306, 2002.

[MR87a] N. V. Murray and E. Rosenthal. Inference with path resolution and semantic graphs. *Journal of the Association of Computing Machinery*, 34(2):225–254, April 1987.

[MR87b] N. V. Murray and E. Rosenthal. Path dissolution: A strongly complete inference rule. In *Proc. of the* 6*th* *National Conference on Artificial Intelligence*, pages 161–166, Seattle, WA, July 12-17 1987.

[MR94] N. V. Murray and E. Rosenthal. On the computational intractabilityof analytic tableau methods. *Bulletin of the IGPL*, 2(2):205–228, 1994.

[MRM$^+$95] G. R. Morrison, Ross, S. M., M. Gopalakrishnan, and J. Casey. The effects of feedback and incentives on achievement in computer-based instruction. In *Contemporary Educationalk Psychology*, volume 20, pages 32–50, 1995.

[MRS01] D. McMath, M. Rozenfeld, and R. Sommer. A computer environment for writing ordinary mathematical proofs. In *LPAR '01: Proc. of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag.

[MRV01] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern-matching compiler. *Electronic Notes in Theoretical Computer Science*, 44(2):161 – 180, 2001. LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).

[MS99a] E. Melis and J. Siekmann. Concepts in proof planning. In *Intellectics and Computational Logic. Papers in Honor of Wolfgang Bibel*, pages 249–264. Kluwer, 1999.

[MS99b] E. Melis and J. H. Siekmann. Knowledge-based proof planning. *Artif. Intell.*, 115(1):65–105, 1999.

[MS05] E. Melis and J. H. Siekmann. e-learning logic and mathematics: What we have and what we need. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! (2)*, pages 639–662. College Publications, 2005.

[MV05] N. Matsuda and K. VanLehn. Advanced geometry tutor: An intelligent tutor that teaches proof-writing with construction. In Chee-Kit Looi, Gordon I. McCalla, Bert Bredeweg, and Joost Breuker, editors, *AIED*, volume 125 of *Frontiers in Artificial Intelligence and Applications*, pages 443–450. IOS Press, 2005.

[MW07] C. Benzmüller M. Wagner, S. Autexier. Plato: A mediator between text-editors and proof assistance systems. In C. Benzmüller S. Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, volume 174(2) of *Electronic Notes on Theoretical Computer Science*, pages 87–107. Elsevier, april 2007.

[Mye96] K. Myers. Advisable planning systems, 1996.

[Ned02] R. Nederpelt. Weak type theory: A formal language for mathematics. Technical report, Eindhoven University of Technology, Department of Math. and Comp. Sc., 2002.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving, 2001.

[NSS57] 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.

[NWE97] N. Nedjah, C. D. Walter, and S. E. Eldridge. Optimal left-to-right pattern-matching automata. In M. Hanus, J. Heering, and K. Meinke, editors, *ALP/HOA*, volume 1298 of *Lecture Notes in Computer Science*, pages 273–286. Springer, 1997.

[OS88] F. Oppacher and E. Suen. Harp: A tableau-based theorem prover. *J. Autom. Reasoning*, 4(1):69–100, 1988.

[OS91] H. J. Ohlbach and J. H. Siekmann. The Markgraf Karl refutation procedure. In Jean Luis Lassez and Gordon Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 41–112. MIT Press, 1991.

[Ott08] J. Otten. leancop 2.0 and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In Alessandro Armando, P. Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 283–291. Springer, 2008.

[Pas01] D. Pastre. Muscadet 2.3: A knowledge-based theorem prover based on natural deduction. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning: Proc. of IJCAR'01*, pages 685–689. Springer, 2001.

[Pas02] D. Pastre. Strong and weak points of the muscadet theorem prover - examples from casc-jc. *AI Commun.*, 15(2-3):147–160, 2002.

[Pau83] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[Pau99] L. C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.

[Pau08] L. C. Paulson. The isabelle reference manual, 2008.

[Pel86] F. J. Pelletier. Thinker. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 701–702. Springer, 1986.

[Pel97] Nicolas Peltier. Simplifying and generalizing formulae in tableaux. pruning the search space and building models. In Galmiche [Gal97], pages 313–327.

[Pel98] F. Jeffry Pelletier. Automated natural deduction in thinker. *Studia Logica*, 60, 1998.

[Pel99] N. Peltier. Pruning the search space and extracting more models in tableaux. *Logic Journal of the IGPL*, 7(2):217–251, 1999.

[Pfe87] F. Pfenning. *Proof Transformation in Higher-Order Logic.* Phd thesis, Carnegie Mellon University, 1987.

[PJ06] H. Passier and J. Jeuring. Feedback in an interactive equation solver. Technical Report UU-CS-2006-021, Department of Information and Computing Sciences, Utrecht University, 2006.

[Pla08] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reasoning*, 41(2):143–189, 2008.

[Pra65] D. Prawitz. *Natural deduction; a proof-theoretical study*, volume 3 of *Stockholm Studies in Philosophy.* Almqvist and Wiksells, 1965.

[PS02] M. Pollet and V. Sorge. Integrating computational properties at the term level. In *Proc. of Calculemus*, pages 78–83, 2002.

[PS06] M. Pollet and V. Sorge. Connecting logical representations and efficient computations. *Electr. Notes Theor. Comput. Sci.*, 151(1):127–142, 2006.

[RBGL07] V. Robinet, G. Bisson, M. B. Gordon, and B. Lemaire. Searching for student intermediate mental steps. In *Complete On-Line Proc. of the Workshop on Data Mining for User Modeling, at the 11th International Conference on User Modeling (UM 2007), Corfu, Greece, June 25, 2007*, pages 101–105, 2007. `http://www.educationaldatamining.org/UM2007/Robinet.pdf`.

[Ric02] J. Richardson. A semantics for proof plans with applications to interactive proof planning. In *Lecture Notes in Computer Science*, 2002.

[RL07] J. Ruan and Z. Lu. Automated generation of readable proofs for a class of limits of sequences and functions. *Symbolic Computation and Education*, 2007.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *JACM*, 12:23–41, 1965.

[Rob00] J. A. Robinson. Proof = guarantee + explanation. In *Intellectics and Computational Logic (to Wolfgang Bibel on the occasion of his 60th birthday)*, pages 277–294, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[Ros00] B. Rosario. Latent semantic indexing: An overview, 2000.

[RR92] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *J. ACM*, 39(2):295–316, 1992.

[RS93] P. D. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.

[RS98] W. Reif and G. Schellhorn. Theorem proving in large theories. In *Proc. FTP 97*, pages 119–124. Kluwer Academic Publishers, 1998.

[RSG98] J. D. C. Richardson, A. Smaill, and I. M. Green. System description: proof planning in higher-order logic with λ-clam. In Claude Kirchner and Hélène Kirchner, editors, *Proc. of the 15th International Conference on Automated Deduction (CADE-98)*, volume 1421 of *LNAI*. Springer, 1998.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and A. Voronkov. Term indexing. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.

[Rud87] P. Rudnicki. Obvious inferences. *J. Autom. Reasoning*, 3(4):383–393, 1987.

[Rüm08] P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2008.

[Rus03] B. Russell. *Principles of Mathematics.* Cambridge University Press, 1903.

[RV99] A. Riazanov and A. Voronkov. Vampire. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[RV01] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.

[SA99] S. Doaitse Swierstra and Pablo R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. *LNCS*, 1725:111–129, 1999.

[Sac10] C. Sacerdoti-Coen. Declarative representation of proof terms. *J. Autom. Reasoning*, 44(1-2):25–52, 2010.

[SB96] B. Sufrin and R. Bornat. User Interfaces for Generic Proof Assistants Part I: Interpreting Gestures. In S. Aitken and P. Gray, editors, *Proc. of User Interfaces for Theorem Provers*, 1996.

[SB98] W. Sieg and J. Byrnes. Normal natural deduction proofs (in classical logic). *Studia Logica*, 60(1):67–106, 1998.

[SB09a] M. Schiller and C. Benzmüller. Granularity-adaptive proof presentation. Technical report, SEKI Working-Paper, 2009.

[SB09b] M. Schiller and C. Benzmüller. Presenting proofs with adapted granularity. In Bärbel Mertsching, M.us Hund, and Zaheer Aziz, editors, *KI 2009: Advances in Artificial Intelligence. KI-09, in Conjunction with 32nd Annual German Conference on AI, Paderborn, Germany, Paderborn, Germany, Germany, Germany*, volume 5803, pages 289–279. Springer Verlag, 2009.

[SB09c] M. Schiller and C. Benzmüller. Proof granularity as an empirical problem? In *Proc. Computer Science in Education (CSEDU)*, pages 350–354. INSTICC Press, 2009.

[SBA06] J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with omega. *Journal of Applied Logic*, 4(4):533–559, 2006.

[SBdV06] M. Schiller, C. Benzmüller, and A. Van de Veire. Judging granularity for automated mathematics teaching. In *LPAR 2006 Short Papers Proceedings*, Phnom Pehn, Cambodia, 2006.

[SBF$^+$03] 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 F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series (28), pages 271–314. Kluwer Academic Publishers, 2003. ISBN 1-4020-1656-5.

[Sch77] K. Schütte. *Proof Theory. (Originaltitel: Beweistheorie)*, volume 255 of *Die Grundlehren der mathematischen Wissenschaften.* Springer, Berlin;Heidelberg;New York, 1977.

[Sch97] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In W. McCune, editor, *Proc. of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 87–100, Berlin, July 13–17 1997. Springer.

[Sch02] S. Schulz. E - a brainiac theorem prover, 2002.

[Sch04] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[Sch05] M. Schiller. Mechanizing Proof Step Evaluation for Mathematics Tutoring - the Case of Granularity. Diploma thesis, Saarland University, Saarbrücken, Germany, 2005.

[Sch06] A. Schairer. *Transformations of Specifications and Proofs to Support an Evolutionary Formal Software Development,*. PhD thesis, Saarland University, 2006.

[Sch09] W. Schreiner. The risc proofnavigator: a proving assistant for program verification in the classroom. *Formal Asp. Comput.*, 21(3):277–291, 2009.

[Sch10] M. Schiller. *Granularity Analysis for Tutoring Mathematical Proofs.* PhD thesis, Saarland University, 2010.

[SDB07] M. Schiller, D. Dietrich, and C. Benzmüller. Towards computer-assisted proof tutoring. In *JEM Workshop on identifying and supporting (scientific) communities in education and research*, Jacobs University Bremen, Germany, 2007.

[SDB09] M. Schiller, D. Dietrich, and C. Benzmüller. Proof step analysis for proof tutoring – a learning approach to granularity. *Teaching Mathematics and Computer Science*, 2009. In print.

[SDM$^+$65] N. A. Shanin, G. V. Davydov, S. Ju. Maslov, G. E. Minc, V. P. Orevkov, and A. O. Slisenko. An algorithm for a machine scan of a natural logical deduction in a propositional calculus. *Translated in (Siekmann and Wrightson 1983)*, pages 424–484, 1965.

[SG89] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(2):101–140, July/August 1989.

[Sie89] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3–4):207–274, March–April 1989.

[Sie09] W. Sieg. Searching for proofs (and uncovering capacities of the mathematical mind). *to appear*, 2009.

[Sla94] J. K. Slaney. Finder: Finite domain enumerator - system description. In A. Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 798–801. Springer, 1994.

[SMS96] J. Silva, P. Marques, and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96: Proc. of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[Smu68] R. M. Smullyan. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik.* Springer-Verlag, Berlin, 1968.

[SN04] R. Sommer and G. Nuckols. A Proof Environment for Teaching Mathematics. *Journal of Automated Reasoning*, 32(3):227–258, 2004.

[Sol05] D. Solow. *How to read and do proofs.* Laurie Rosatone, 2005.

[Sor00] V. Sorge. Non-trivial symbolic computations in proof planning. In H. Kirchner and C. Ringeissen, editors, *FroCos*, volume 1794 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2000.

[SRA99] P. G. Fairweather Sherman R. Alpert, M. K. Singley. Deploying intelligent tutors on the web: An architecture and an example. *Int. J. Artif. Intell. Ed.*, 10(2):183–197, 1999.

[SRL$^+$06] W. Sieg, J. Ramsey, D. Lafon, D. McLaughlin, T. Gibson, and D. Perkins. Apros, 2006.

[SS94] W. Sieg and R. Scheines. Computer Environments for Proof Construction. *Interactive Learning Environments*, 4(2):159–169, 1994.

[ST89] P. Suppes and S. Takahashi. An interactive calculus theorem-prover for continuity properties. *J. Symb. Comput.*, 7(6):573–590, 1989.

[Sta78] R. Statman. Bounds for proof search and speed-up in the predicate calculus. *Annals of Mathematical Logic*, 15:225–287, 1978.

[Sti90] Mark E. Stickel, editor. *Proc. CADE-10*, volume 449 of *LNAI*. Springer Verlag, July 1990.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[SW83] Jörg Siekmann and Graham Wrightson, editors. *Automation of Reasoning 1: Classical Papers on Computational Logic 1957–1966*, Symbolic Computation. Springer Verlag, 1983.

[Sym97] D. Syme. DECLARE: a prototype declarative proof system for higher order logic. Technical Report UCAM-CL-TR-416, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, February 1997.

[Sym99] D. Syme. Three tactic theorem proving. In *Theorem Proving in Higher Order Logics, TPHOLs '99*, pages 203–220. Springer, 1999.

[Tai79] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[TB85] A. Trybulec and H. A. Blair. Computer assisted reasoning with mizar. In *IJCAI*, pages 26–28, 1985.

[TCK95] L. Thery, Y. Coscoy, and G. Kahn. Extracting text from proofs. In *Typed Lambda Calculus and its Applications*, pages 109–123. Springer-Verlag, 1995.

[Tur37] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230–265, 1937. **43**:544-546.

[Urb06] J. Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[Urq98] A. Urquhart. The complexity of propositional proofs. *Bulletin of the EATCS*, 64, 1998.

[VB98] E. Visser and Z. Benaissa. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[VBA03] Q. B. Vo, C. Benzmüller, and S. Autexier. Assertion application in theorem proving and proof planning. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 0–127. Kaufmann, 2003.

[vdBHKO02] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[vdBKV03] M. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[vdH01] J. van der Hoeven. Gnu TEXMACS: A free, structured, wysiwyg and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[Vis01] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In Harald Ganzinger, editor, *RTA*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–167. Springer, 1996.

[VLP07] K. Verchinine, A. V. Lyaletski, and A. Paskevich. System for automated deduction (sad): A tool for proof verification. In F. Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 398–403. Springer, 2007.

[vS00] B. von Sydow. Alfie, a proof editor for propositional logic, 2000.

[WAB$^+$99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th Conference on Automated Deduction*, number 1632 in LNAI, pages 378–382. Springer, 1999.

[Wac05a] B. Wack. *Supernatural Deduction. Nancy 1, France. E-mail.* PhD thesis, LORIA & Université Henri Poincaré, 2005.

[Wac05b] B. Wack. *Typage et déduction dans le calcul de réécriture.* Thèse de doctorat, Université Henri Poincaré (Nancy 1), octobre 2005.

[Wag10] M. Wagner. *Change-Oriented Architecture for Mathematical Authoring Assistance.* PhD thesis, FR 6.2 Informatik, Universität des Saarlandes, 2010.

[Wal90] L. Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics.* MIT Press series in artificial intelligence, 1990.

[War84] D. S. Warren. Efficient prolog memory management for flexible control strategies. In *Proc. of the 1984 Int. Symps. on Logic Programming*, pages 198–202, 1984.

[WBH$^+$09] M. Wolska, M. Buckley, H. Horacek, I. Kruijff-Korbayova, and M. Pinkal. Linguistic processing in a mathematics tutoring system: Cooperative input interpretation and dialogue modelling. In Matthew W. Crocker and J. Siekmann, editors, *Resource-Adaptive Cognitive Processes*, Cognitive Technologies, pages 267–289. Springer Berlin Heidelberg, 2009.

[WBPB01] C. Webber, L. Bergia, S. Pesty, and N. Balacheff. The baghera project: a multi-agent architecture for human learning. In *Workshop - Multi-Agent Architectures for Distributed Learning Environments. Proc. International Conference on AI and Education*, pages 12–17, 2001.

[Wen99a] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Bertot et al. [BDH$^+$99], pages 167–184.

[Wen99b] M. Wenzel. Miscellaneous isabelle/isar examples for higher-order logic, 1999.

[Wie04] F. Wiedijk. Formal proof sketches. In Mario Coppo Stefano Berardi and Ferruccio Damiani, editors, *Types for Proofs and Programs: Third International Workshop TYPES 2003*, number 3085 in LNCS, pages 378–393, Torino, 2004. Springer.

[Wie05] F. Wiedijk. Nine formal proof sketches. 2005.

[Wil93] D. E. Wilkins. *Using the SIPE-2 planning System: A Manual for Version 4.3*, 1993.

[Win06] W. Windsteiger. An automated prover for zermelo-fraenkel set theory in *heorema*. *J. Symb. Comput.*, 41(3-4):435–470, 2006.

[Wir04] C.-P. Wirth. Descente infinie + deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004.

[Wos90] L. Wos. The problem of finding a mapping between clause representation and natural-deduction representation. *J. Autom. Reasoning*, 6(2):211–212, 1990.

[WR10] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910.

[YBG$^+$94] T. Yoshida, A. Bundy, I. Green, T. Walsh, and D. A. Basin. Coloured rippling: An extension of a theorem proving heuristic. In *ECAI*, pages 85–89, 1994.

[Zam99] V. Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [BDH$^+$99], pages 185–202.

[Zer08] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. *Mathematische Annalen*, 65:261–281, 1908.

[Zha96] J. Zhang. Constructing finite algebras with falcon. *J. Autom. Reasoning*, 17(1):1–22, 1996.

[Zha97] H. Zhang. Sato: An efficient propositional prover. In W. McCune, editor, *Proc. of the 14th Conference on Automated Deduction*, number 1249 in LNAI. Springer, 1997.

[Zin06] C. Zinn. Supporting tutorial feedback to student help requests and errors in symbolic differentiation. In Mitsuru Ikeda, Kevin D. Ashley, and Tak-Wai Chan, editors, *Intelligent Tutoring Systems*, volume 4053 of *Lecture Notes in Computer Science*, pages 349–359. Springer, 2006.

[ZM04] J. Zimmer and E. Melis. Constraint solving for proof planning. *Journal of Automated Reasoning*, 33(1):51–88, July 2004.

[ZZ95] J. Zhang and H. Zhang. Sem: a system for enumerating models. In *IJCAI*, pages 298–303, 1995.