Knowledge-Based Proof Planning

Erica Melis Jörg Siekmann
Universität des Saarlandes, Fachbereich Informatik and DFKI
D-66041 Saarbrücken, Germany
email:{melis,siekmann}@cs.uni-sb.de

Abstract

Knowledge-based proof planning is a new paradigm in automated theorem proving (ATP) which swings the motivational pendulum back to its AI-origins in that it employs and further develops many AI-principles and techniques such as hierarchical planning, knowledge representation in frames and control-rules, constraint solving, tactical and meta-level reasoning. It differs from traditional search-based techniques in ATP not least with respect to its level of abstraction: the proof of a theorem is planned at an abstract level and an outline of the proof is found. This outline, i.e. the abstract proof plan, can be recursively expanded and it will thus construct a proof within a logical calculus. The plan operators represent mathematical techniques familiar to a working mathematician. While the knowledge of a domain is specific to the mathematical field, the representational techniques and reasoning procedures are general-purpose. The general-purpose planner makes use of this mathematical domain knowledge and of the guidance provided by declaratively represented control-rules which correspond to mathematical intuition about how to prove a theorem in a particular situation. These rules provide a basis for meta-level reasoning and goal-directed behaviour. We demonstrate our approach for the mathematical domain of limit theorems, which was proposed as a challenge to automated theorem proving by the late Woody Bledsoe. Using the proof planner of the ΩMEGA system we were able to solve all the well known challenge theorems including those that cannot be solved by any of the existing traditional systems.

Keywords: theorem proving, planning, automated proof planning, meta-level reasoning, integrating constraint solvers

Contents

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

Woody Bledsoe, 1986

1 Introduction

Since the early days of artificial intelligence (AI) research, two schools have existed in automated theorem proving, the logic-oriented approaches and the rather psychologically oriented approaches which try to simulate people. For example, at the Dartmouth Conference in 1956, two systems found wide attention and are considered 'classic' today: Martin Davis' decision procedure based on Presburger's Arithmetic [33] which is the grand ancestor of logic-oriented systems and the now seminal Logic Theorist [98] that pioneered the second category. In 1954, Davis' system was the first system ever to prove a theorem with a computer ("The sum of two even numbers is even.") and about a year later, Alan Newell and Herb Simon finished their joint system, later called the Logic Theorist, which succeded in proving many theorems from Principia Mathematica [107] and sparked off the field of artificial intelligence.

Since Hao Wang's work [121] and with the development of the resolution principle in 1965 [105], the logic-oriented, search-based paradigm has by and large dominated the field, and by far the strongest systems were built within this train of thought.

Other ideas were, however, never fully absent. Woody Bledsoe, among others [7, 21, 104], advocated automated theorem proving based on mathematical knowledge and practice [15]. Bledsoe's beautiful quotation above shows his scepticism for purely search-based theorem proving, and in fact he never believed that it would succeed in proving difficult theorems even in mathematically well-understood domains. He developed a vision of the concepts and procedures necessary for automated theorem proving since he thought of himself as "one of the researchers working on resolution type systems who 'made the switch'... and became convinced that we were on the wrong track" [14].

Traditional automated theorem proving is based on general-purpose machine-oriented logical calculi such as the resolution calculus [105], the tableaux- [115], or the matrix methods [5, 13]. The inference rules of such a calculus span the search space and more than thirty years of research led to a battery of refinements and strategies to traverse these large (billions of nodes) spaces. In spite of many attempts to the contrary (e.g., the hyperresolution rule), the inference steps defined by these calculi are rather small and low-level when compared with the proof steps of a trained mathematician.

Traditional systems such as the MKRP system [36], OTTER [81], SETHEO [73], or SPASS [27] are essentially black boxes; after the input of a theorem and of (hopefully) exactly those axioms necessary for the proof of the theorem and after setting the appropriate parameters, the system searches blindly for a sequence of logic rules that proves the theorem from the axioms. The search is supported by some general-purpose control, called strategies or refinements [76], that is purely syntactic in nature and hardly reflects mathematical ways of discovering a proof.

Now this approach, although far from any mathematical practice and often under attack from the more AI-oriented community [47, 48], is not entirely unreasonable as, e.g., the chess program Deep Blue has demonstrated which also derives its strength from search techniques. Recent success with blind SAT-techniques seems to corroborate even more the advantages of blind but fast and simple mechanisms over domain-dependent AI-techniques.

Just like automated chess and other areas, traditional ATP systems benefit from the technological development of faster computers with larger storage – they can now store and search billions of clauses and indeed, they do.[1] Due to this improvement and to various technical advances in representational techniques (see for indexing [44]), systems have gained considerable strength and they can prove non-trivial open mathematical problems, such as the Robbins Algebra Conjecture [80], whose proofs are often unintuitive and therefore tricky for humans. In general, however, most proofs of genuinely mathematical problems even in well-understood domains are very much beyond the capabilities of any of today's systems. So, after forty years of research the time is ripe again to ask Woody Bledsoe's question: are we on the wrong track?
It appears that this situation is not unique just for automated theorem proving

> *Over time we become trapped in our shared vision of appropriate ways to tackle problems, and even more trapped by our funding sources where we must constantly justify ourselves by making incremental progress. Sometimes it is worthwhile stepping back and taking an entirely new (or perhaps very old) look at some problems and to think about solving them in new ways. This takes courage as we may be leading ourselves into different sorts of solutions that will for many years have poorer performance than existing solutions. With years of perseverance we may be able to overcome initial problems with the new approaches and eventually leapfrog to better performance. Or we may turn out to be totally wrong. That is where the courage comes in.* (Rodney Brooks AAAI-96)

The Psychology of Mathematical Invention

Why can a mathematician cope with long and complex proofs and what are her strategies for avoiding less promising proof paths?

Given the current state of knowledge about our human formal reasoning capacity, we do not know the answer. However, at least the following three observations are generally accepted: first, a mathematician's reasoning is more often than not based on some vivid and concrete representation of the problem at hand [45, 119]. Secondly, difficult theorems are not

[1]The first theorem prover implemented by the second author in the mid seventies was not untypical for its generation; it could search spaces of several 100,000 clauses, to generate proofs of the length of a dozen steps. Our MKRP-system that was under development for almost fifteen years could search spaces of several million clauses by the end of the eighties to find proofs of about a hundred steps. Todays systems, such as SPASS and OTTER search spaces well in the billions: "Clauses generated: about 3,000,000,000. At the end of the search, about 400,000 clauses were in use. This took 314 hours and used 433 Megabyte of RAM. I guess, for a successful search the search space would be 1/4 as big." Bill McCune on OTTER's latest figures for large search spaces for an open problem in combinatory logic. Personal communication.

shown from scratch but usually with some known proof technique such as proof by induction, the pigeon hole principle, proof by diagonalization, and so on. In fact, a good mathematician has at least several dozen (but presumably less than a thousand) proof techniques for her particular field at her disposal. Also, there surely are thousands (but probably less than a million) minor tricks of the trade such as when and how to apply a homomorphism, when to differentiate, or how to reformulate a given problem. In hindsight it seems preposterous to assume that all of this can be achieved just by blind search. For very difficult and open mathematical problems – usually they are open exactly because none of the standard attacks yields a solution – there is the need to combine and reshuffle these standard techniques and so, thirdly and finally, there is empirical evidence [102, 67] which suggests that mathematicians *plan* a proof at various levels of abstraction in the proof discovery process. The following quotation taken from an interview with the German mathematician Faltings, who proved Mordell's Conjecture, beautifully illustrates the case in point.[2] When asked how he proved the famous problem he said [37]

> *Man hat Erfahrungen, dass bestimmte Schlüsse unter bestimmten Voraussetzungen funktionieren... Man überlegt sich also im Groben: Wenn ich das habe, könnte ich das zeigen und das nächste. Hinterher muss man die Details einfügen und sieht, ob man es auch wirklich so machen kann.*

To translate freely into English, "We know from experience that certain inferences are usually successful under certain prerequisites. So first we ponder about any reasonable way how to proceed. In other words, we roughly plan; if we would have a certain result the next result may follow and then the next etc. Afterwards we have to fill in the details, and to check whether the plan really works."

Proof plans seem to have a cognitive reality of their own and in the following two sections we like to mention two areas of research that also benefit from the new way of seeing things, i.e., from the fact that there is a well defined concrete representational level above the level of a logical calculus.

Proofs by Analogy

Theorem proving by analogy is a small subarea of ATP which has been plagued by a notorious problem, namely that we often find two proofs analogous although it is hard to establish a one to one mapping from the syntactic representation of the source proof to the target – and hence, the techniques based upon such mappings often fail (see, e.g., [82]). In particular, the *proof idea* may be the same while the details of source and target proof may differ considerably. For instance, not every detail of the proof of LIM+ (the limit of sum of two functions equals the sum of their limits) can be transferred to a proof of LIM* (the limit of the product of two functions equals the product of their limits) although, surely, the theorems are very similar and a mapping between them can be established. Furthermore,

[2] In the early 80s Gerd Faltings solved Mordell's Conjecture which says "Algebraic curves of order 2 or more have finitely many rational points." Mordell's Conjecture has been considered a hard mathematical problem and it took over 60 years to solve it. Last but not least it derives its significance from being a close relative of Fermat's famous conjecture. Faltings received the field medal for this work.

applying the same inference rule in the source and target situations does not necessarily lead to subgoals that match, e.g., even if the multiplication function is identically mapped, the procedure decomposing a natural number into a product of primes may lead to different numbers of factors in source and target.

So, how do we fix the right level of abstraction, if it is not the calculus-level? Proof *plans* are more abstract than the calculus-level proofs and often capture the "essential idea" of a proof explicitely. In [83] we have shown how this new representation leads to a much improved analogical transfer using *derivational analogy* as introduced by Jaime Carbonell [26]. Analogy-driven proof plan construction [83] was first implemented for proofs by mathematical induction [94] and later also used to solve an open challenge problem of Woody Bledsoe in [85].

Proof Presentation

The output of a successful run of a traditional system is notoriously unreadable let alone intuitively understandable in the way mathematicians communicate. Take, for instance, the following theorem.
Theorem: Let K be an ordered field. If $a \in K$, then $1 < a$ implies $0 < a^{-1} < 1$ (and vice-versa).

The proof generated by the prover OTTER, looks as follows.

```
1 [] x=x.
2 [] -(x<y)| -(0<z)|x*z<y*z.
3 [] -(x<y)| -(y<z)|x<z.
4 [] -(x<y)| -(y<x).
5 [] x=0| -(0<x)|0<inv(x).
7 [factor,4,1,2] -(x<x).
8 [] -(0<inv(a))| -(inv(a)<1).
9 [] 0<1.
10 [] x=0|x*inv(x)=1.
12,11 [] 1*x=x.
13 [] 1<a.
19 [hyper,3,9,13] 0<a.
27 [para_into,19.1.2,5.1.1,unit_del,7,19] 0<inv(a).
48,47 [para_from,10.1.1,19.1.2,unit_del,7] a*inv(a)=1.
60 [hyper,2,13,27,demod,12,48] inv(a)<1.
65 [hyper,8,27,60] F.
```

while the textbook proof taken from [77] reads
Proof: Let $1 < a$. According to lemma 1.10 we have $a^{-1} > 0$. Therefore $a^{-1} = 1a^{-1} < aa^{-1} = 1$. qed.

This huge discrepancy is more than the annoying technical problem of translating a machine found proof into natural language (see [1]). Obviously the human readable form of the above proof is at a very different level of abstraction and gives an "outline", where the details and possibly a translation into a calculus-level proof could in principle be provided by an experienced mathematician.

In addition, for more difficult proofs, a proof in mathematics is both, a means to understand and communicate why some result holds and secondly a means to achieve precision [78]. Alan Robinson coined the formula [105]

$$\text{Proof} = \text{Guarantee} + \text{Explanation}.$$

As to a 'guarantee', we have the proof in the logical calculus that could be checked by a proof checking program, however, it is a logician's folly to assume that this is also an explanation. The logical proof provides a 'justification', rather than an 'explanation'. Alan Robinson [106] suggests that the proof-as-explanation aspect is both (far) more important and (far) more interesting than the mere logical guarantee.

For a *comprehensible* communication, an explanation at a more *abstract* level is required. Such an explanation can, however, not be generated directly from a calculus-level proof. We shall argue that the presentation of a textbook proof is based not on the calculus-level expansion but, depending on the intended reader, on a proof plan at one of the possible levels of abstraction.

An Alternative: Proof Planning

Proof planning was originally conceived as a mere extension of tactical theorem proving (see LCF [43], NuPrl [31], or Isabelle [100] for tactical theorem proving). Tactical theorem proving is based on the notion of a *tactic* which encapsulates repeatedly occurring sequences of inference steps into macro-steps. These tactics are realized by programs, i.e., executing a tactic yields a sequence of calculus-level proof steps and thus relieves the user from applying too many single inference rules in a row in interactive theorem proving.

The idea of proof planning is as follows. 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. Now if an individual tactic of this kind, which may also be a call of a traditional automated theorem prover, a fast decision procedure, or a computer algebra system, is augmented by pre- and postconditions, we can *plan* such a sequence of tactics. This marriage of planning operators with tactics was Alan Bundy's key idea [22] for proof planning.

More recently, another type of methods with schematic expansions, has been defined [56]. Hence, methods are not necessarily restricted to its tactical origins but can be defined more generally as proof planning operators [93], i.e. any building blocks for proof plans, that can be recursively expanded into a calculus-level proof.

Proof planning searches for a plan, i.e., for a sequence of methods, where the precondition of a method *matches* the postcondition of its predecessor in that sequence. This well known view of a plan [38] leads naturally to a new engine for automated theorem proving; a planner can be used to *plan* a sequence of methods that transforms the proof assumptions into the theorem. Standard heuristics and techniques from the planning literature can now be employed.

Moreover, proof planning provides means of *global search control* that correspond well to mathematical intuition, as opposed to the local and syntactic search heuristics which are commonly used for search control in traditional automated theorem proving [76].

The first proof planner, *CLAM*[23], was designed to prove theorems by mathematical induction. Its proof steps are abstract in the sense that they ignore the costly type checking required in the underlying Martin-Löf-type theory. *CLAM* employs the rippling search heuristic for difference reduction [57, 24]. Rippling is based on an annotated logic calculus that handles annotated terms and uses annotated matching. More specifically, a skeleton-annotation indicates the commonalities between the induction hypothesis and the induction conclusion (the 'skeleton') and a context-annotation indicates the difference between the induction hypothesis and the induction conclusion (the 'context'), and rippling is essentially a difference-reducing rewriting technique that preserves the skeleton, i.e., the commonalities. Such a difference reduction works in particular for equational proofs and proofs by mathematical induction. However, there are many theorems that are difficult or impossible to prove by *CLAM* (for instance, the limit theorem LIM*) because (i) for many problems, the means of control are insufficient and not flexible enough, (ii) no domain-specific methods are used, and (iii) the need to construct mathematical objects with certain theory-specific properties is not supported.

To extend proof planning, this article introduces *knowledge-based proof planning*. First it shows which knowledge is available in mathematics and how it can be represented. Section three describes how a general-purpose proof planner makes use of mathematical domain knowledge including methods, control-rules, and external reasoners. Several techniques that restrict the search in proof planning are introduced, in particular *meta-level reasoning* in §3.2 and *constraint solving* in §3.3.3. In §4 we show how we proved limit theorems using the knowledge-based proof planner of the ΩMEGA system [12] and thereby show that knowledge-based proof planning is not only possible in principle but can be used to solve problems that cannot be solved by other general-purpose systems. Finally, §5 argues that proof plans provide a representational abstraction of a proof that is also well-suited for the communication with a user.

ΩMEGA is a complex theorem proving system for interactive and automated proof development whose distributed architecture integrates various (mathematical) services, such as the proof planner, traditional automated theorem provers, the knowledge base, computer algebra systems, a proof verbalization component, and a graphical user interface. In the following, we shall restrict our presentation of ΩMEGA to those aspects that are directly relevant for knowledge-based proof planning, the ΩMEGA system itself is documented inter alia in [12, 111, 112].

2 Principles of Proof Planning

We shall now briefly review the basic notions from the field of planning in AI and subsequently introduce proof planning within this terminology. Our extensions that led to knowledge-based proof planning are then presented in §3. We use σ for substitutions and abbreviate the result of applying σ to F by $F\sigma$.

2.1 Basics of Planning

A planning problem consists of an *initial state* describing some initial situation and of *goals* to be achieved. A planning domain is defined by *operators* that usually represent actions. The operators have specifications to be used in the planning process. In STRIPS notation [38] these specifications are called preconditions and effects. Preconditions specify the conditions of the planning state that have to be satisfied for the operator to be applicable, whereas effects describe the potential changes of the planning state caused by an operator application. In STRIPS representation, effects are represented by add-lists ($\oplus$) and delete-lists ($\ominus$), i.e. lists of literals that are added or deleted by an operator application. Note that preconditions and effects are usually formulated as expressions in a restricted first-order logical *object-level* language such as `(on A B)` or `(arm-holds X)`. However, some planners, e.g. Prodigy [97], allow for additional preconditions formulated in a *meta-level* language which restrict the instantiation of parameters. They are called application conditions in the following.

A partial plan is a partially ordered set of steps, i.e. of (partially) instantiated operators, with additional instantiation constraints and auxiliary constraints [63]. A partial plan can be seen as an implicit representation of a set of sequences (set of potential solutions) consistent with the ordering, instantiation, and auxiliary constraints. A solution of a planning problem, a complete plan, is a fully instantiated linearization of a partial plan that transforms the initial state into a goal state, i.e., a state in which the goals hold.

The operation of a planner repeatedly refines a partial plan, i.e., it adds steps and constraints and thus restricts its set of potential solutions until a solution can be picked from its set of potential solutions [62]. Table 1 shows a simplified backward planning algorithm (not handling goal interactions). Planning starts with a partial plan π_0 defined by the problem to be solved, where π_0 consists of steps t_0 and t_∞ that are instantiations of the dummy operators `start` and `finish`. They have the initial state as $\oplus$-effects and the goals as preconditions, respectively. π_0 also represents the order constraint $t_0 \prec t_\infty$. The introduction of a step into a partial plan removes an open goal g from the goal agenda G and may introduce new open subgoals and constraints. This refinement is continued until no open goals are left and a solution is found or until no operator is applicable anymore.

Search is involved in each of the planning algorithms, and hence, the process of planning consists of a sequence of choices that leads to a complete plan. These decisions include to choose which open goal to solve next, which planning operator to use in order to attain this goal, and which instantiations to choose. These decisions influence the way the search space is traversed or restricted. Some planners use explicit declarative control-rules to guide the search.

A diversity of planning approaches has been developed, among them two kinds of hierarchical planning techniques, *precondition abstraction* [108] and *operator abstraction* planning [117], also called hierarchical task network (HTN)-planning. Precondition abstraction first searches in an abstract space by ignoring certain preconditions of operators. These ignored goals are considered later at a lower hierarchical level of planning only. Operator abstraction employs complex (as opposed to primitive) operators that represent complex actions. A complex operator can be *expanded* to a partial (sub)plan according to a schema. Since only primitive actions can be executed, all complex operators have to be expanded in

`Backwards-Refine-Plan`(π, G)
Termination: **if** goal agenda G empty, **then** `Solution`. **if** no operator applicable, **then** `Fail`. **Goal Selection**: Choose open goal $g \in G$. **Operator Selection**: • For each operator M for each $\oplus$-effect e of M let σ be the matcher $e\sigma = g$ **if** application-conditions$(M\sigma) = true$, **then** M is applicable. • Choose one applicable M (backtracking point) – insert M into π – insert constraints into π – update G. **Recursion**: Call `Backwards-Refine-Plan` on the refined partial plan π.

Table 1: Outline for backward planning

order to obtain an executable plan.

After this brief review of classical AI-planning we shall now describe how this transfers to proof planning. Search problems in proof planning and extensions of proof planning to cope with the large search space are addressed afterwards in section 3.

2.2 Proof Planning

Proof planning cosiders a proof problem as a planning problem and hence it is an application of planning techniques to mathematics. This domain exhibits seriously complex problems and potentially infinitely branching search spaces. However, the notorious problem of goal interaction is typically not present in this domain. The particular planning algorithm we are actually using is not really important for a general article as this (in ΩMEGA we experiment with two different algorithms) hence we just refer to a simple proof planner for backward and forward planning, extended by hierarchical planning.

The initial state in proof planning is a collection of sequents,[3] the proof assumptions, and the goal is the sequent to be proven. A proof planning domain consists of *methods*, of *control-rules*, and *theory-specific reasoners*. A *partial proof plan* is a partially ordered set of instantiated methods and a *complete*, i.e., fully expanded proof plan is a sequence

[3]A sequent is an object $(\Delta \vdash F)$ with a set of formulae Δ and a formula F which means that F is derived from Δ.

of instantiated primitive methods that transfers the initial state into a goal state. Thus a complete plan is a solution of a problem, in other words, a proof of the theorem.

The beauty of this approach is that methods represent familiar mathematical proof techniques or frequently occurring mathematical formula manipulations. In particular, methods should capture

1. common patterns in proofs, that is, a common *structure* such as in proofs by mathematical induction (see Figure 1) or

2. common proof *procedures* such as term simplifications or, for example, the application of the Hauptsatz of Number Theory (each natural number can be uniquely represented as the product of prime numbers).

Induction
- Base-case
  - Symbolic evaluation
  - Simplification
  - Tautology checking
- Step-case
  - Rewriting
  - Fertilization

Figure 1: Proof pattern for mathematical induction, where fertilization is the application of the induction hypothesis

Examples for methods that encode a common proof structure are `Diagonalization` [28], `Induction` [23], and the theory-specific `ComplexEstimate` [86] which is an estimation method used for planning limit theorems as presented in § 4. Methods that encode common procedures, i.e., methods that have some control encoded, are among others `ComputeIntegral`, `Optimise` [111], or `SimplifyTerm`.

The intuitive mathematical counterpart of methods is reflected in good textbooks. For instance, the first chapter of the textbook *Elements of the Theory of Computation* [74], by Lewis and Papadimitriou introduces the common proof techniques of Mathematical Induction, the Pigeonhole Principle, and the Diagonalization Principle as the main tools to be used throughout the book. The textbook *Computability, Complexity, and Languages* [34] describes the following common pattern in diagonalization proofs (see Figure 2).

- A certain set E is enumerated in a suitable fashion

- It is possible, with the help of the enumeration, to define an object d that is different from every object in the enumeration.

- The definition of d is such that d must belong to E, contradicting the assertion that we began with an enumeration of *all* the elements in E.

Find enumeration of E

Define object d

d different from every e in E

and d in E

Derive contradiction

Figure 2: Proof pattern of diagonalization proofs.

From this description, a planning method for diagonalization proofs was used to prove well known and mathematically difficult theorems by diagonalization [28].

Let us now make these general ideas more concrete. The initial state in proof planning is a collection of proof assumptions formalized by logical sequents of an object-level language[4] and the goal is the sequent to be proven. For instance, for proving the theorem LIM+ the goal is

$$\emptyset \vdash \lim_{x \to a} f(x) + g(x) = L_1 + L_2 \tag{1}$$

and the initial state consists of the proof assumptions

$\emptyset \vdash \lim_{x \to a} f(x) = L_1$ and

$\emptyset \vdash \lim_{x \to a} g(x) = L_2$

and of axioms and definitions of the theory $\mathbb{R}$ of real numbers.

Proof planning now starts with the partial plan π_0 defined by the proof assumptions and the theorem to be proven and searches for a solution of the problem, i.e., a sequence of instantiated methods that transforms the initial state into a goal state. ΩMEGA 's planner searches by backward planning from the goal *and* by goal-oriented forward planning from the assumptions.

ΩMEGA uses both hierarchical planning techniques, operator abstraction and precondition abstraction as introduced above. As a generalization of operator abstraction planning, complex methods can be expanded *recursively* into primitive methods that represent inference steps at the calculus-level of a natural deduction calculus. Each of these expansions is stored in the hierarchically organized proof plan data structure (PDS) [12] as shown in Figure 3. A PDS is used to represent the various levels of proof abstraction; the initial PDS consists of the initial partial plan π_0 and as π_0 is refined by planning, more nodes are introduced into the PDS. The expansion of nodes takes a method and expands it into a subplan at the next lower level of abstraction.

In the figure, three abstraction levels are depicted. The left hand side of Figure 3 sketches the correspondence between calculus-level rules that are composed into schemata or combined

[4] ΩMEGA's current object-level language, POST, is based on Church's simply typed λ-calculus [29].

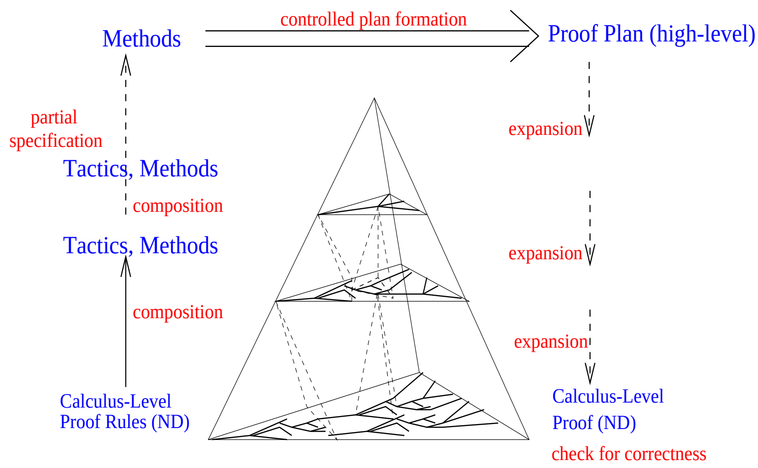

Figure 3: Proof plan data structure with expansions

by tactics as specified by methods. The expansion of high-level methods into lower-level plans until finally the level of the ND-calculus is reached, is indicated on the right hand side. A fully expanded plan, i.e., an ND proof, can be checked for correctness by a proof checker at the end of the planning process. This is necessary because not every instantiated method is guaranteed to be correct in the sense that it yields a correct proof of the conclusion from the premises.

Depending on whether a method encodes a particular proof *structure* or a proof *procedure*, the expansion of a non-primitive method is realized schematically or procedurally. A schematic expansion defines a proof tree that may contain meta-variables.[5]

3 Knowledge-Based Proof Planning

Mathematicians have accumulated mathematical problem solving knowledge over hundreds of years and a mathematician is an expert in a highly specialized mathematical field rather than a universal expert. That is, domain-specific knowledge is a key for success while, of course, some general techniques and mathematical knowledge are important too.

In this section, we shall see which knowledge is available, how it can be represented, how it can be used, and how the architecture of a system makes this knowledge accessible for proof planning.

3.1 Methods

Woody Bledsoe remarks in [16] that central in mathematical knowledge are "... methods, procedures, and tricks of the trade, which have been used successfully by the great mathematicians over the years, also diagrams, constructions, figures, and examples". They play

[5]Meta-variables are place holders for syntactic objects of the underlying logical calculus, i.e. for first-order formulae.

a key role in proof discovery. Accordingly, the design of methods which capture these techniques is essential also for successful proof planning.

Now there exist theory-dependent and theory-independent methods; for instance, decomposition tricks to be used for estimations in ϵ-δ-proofs or the computation of integrals can be considered theory-specific (for the real numbers) and also proof by induction or the abstract consistency property presuppose some minimal mathematical structure and content. On the other hand, a case-split is theory-independent and therefore corresponds to primitive theory-independent methods usually captured in natural deduction calculi [41, 103].

How can we discover methods? One heuristic for knowledge acquisition not only from mathematical (text)books is the following: The importance of a method is more often than not indicated by *naming* it. Named mathematical methods are, for instance, a proof by diagonalization or by induction, the application of an important theorem such as the Hauptsatz of Linear Algebra (each n-dimensional vector space has a basis of n linearly independent vectors), the Hauptsatz of Number Theory, etc. The mathematical monograph *Introduction to Real Analysis* [9] introduces mathematical methods by referring, for example, to the Supremum Property, to the Monotone Convergence Theorem, etc. It states as a didactic help or hint for the reader:"The next two results will be used later as methods of proof"(p.32), "We shall make frequent and essential use of this property (the Supremum Property of $\mathcal{R}$)" (p.45), or "the method we introduce in this section (associated with the Monotone Convergence Theorem) ... applies to sequences that are monotone..." (p.88).

How can methods be represented? Methods consist of declarative specifications to be used in the planning process and of an expansion function that realizes an expansion of the method into a partial proof plan. We distinguish object-level and meta-level specifications. The object-level specifications correspond to the usual preconditions and effects in planning. That is, they specify the sequents that match open goals to be satisfied by the method (backward planning), the sequents that have to be in the planning state when the method is applied (i.e. the subgoals produced by the method during backward planning), the sequents that match with assumptions in the planning state, and those that are produced as new assumptions when the method is used in forward planning.

The meta-level specifications capture in a meta-language the *local* conditions of the method's application, i.e., properties and relations that must hold for the sequents to be processed by the method.[6] The meta-level specification of a method expresses *legal* conditions for the method's application, in particular, restrictions of the instantiation of the method's parameters.

In ΩMEGA methods are represented as frame-like data structures with slots, slot names, and fillers. The slot names are *premises*, *conclusions*, *application conditions*, and *proof schema*. The *premises* and *conclusions* constitute a logical specification, in the sense that the conclusions are supposed to follow logically from the premises. Their ($\oplus$)- and ($\ominus$)-annotations indicate object-level specifications, with the semantics that a $\ominus$-conclusion is deleted as an open goal, a $\oplus$-premise is added as a new open goal, a $\ominus$-premise is deleted as an assumption, and a $\oplus$-conclusion is added as an assumption when the methods are inserted into the partial plan. These specifications are matched with the current goals and

[6] E.g., $subform(G, A)$ expresses the fact that G is a subformula of A, where $\Delta \vdash G$ is a *conclusion* and $\Delta \vdash A$ is a *premise*.

assumptions respectively and then the output of the method's application is computed from the instantiated specifications.

For example, the method `PeanoInduction` below has the object-level specification $\oplus$L1, $\oplus$L2, and $\ominus$L3, where $\oplus$L1 is an abbreviation for the sequent in proof line L1 in the *proof schema*. The annotations mean, the sequent in L3 is deleted as a goal and the sequents in L1 and L2 are added as new subgoals.

method: `PeanoInduction`			
premises	$\oplus$L1, $\oplus$L2		
conclusions	$\ominus$L3		
appl.cond	sort(n) = Nat		
proof schema	L1.	$\vdash P(0)$	(baseCase)
	L2.	$\vdash P(k) \rightarrow P(k+1)$	(stepCase)
	L3.	$\vdash \forall n.P(n)$	(IndAxiom; L1,L2)

This representation of preconditions and effects is somewhat more involved than the usual object-level preconditions and effects because we have forward planning (planning new assumptions from existing ones) as well as backward planning (reducing a goal to subgoals) and, therefore, assumptions and open goals are both included into the planning state: The potential changes of the goals *and* the assumptions have to be represented.

The *application conditions* (*appl.cond*) are formulated in a meta-language using decidable meta-predicates. They specify *local* and *legal* conditions for the method's application. For instance, in the above `PeanoInduction`, the *application condition* requires that n is a natural number.

The slot *proof schema* provides the information for the schematic expansion of the method and thereby captures the semantics of this method, i.e., the schematic expansion introduces a partial plan into the PDS that is defined by this *proof-schema*. The justifications for each line, the right most entries of the proof lines, can be methods again (including ND-rules), or a variable, a call to an external reasoning system, or OPEN. For instance, in the line

L1. $\quad \vdash P(0) \quad$ (baseCase)

of `PeanoInduction` the line-justification is the method `baseCase` that is supposed to prove P for the number 0. A line like

L. $\Delta \quad \vdash |k| \leq \mathbf{M} \quad$ (OPEN)

(from the `ComplexEstimate` method below) means that line L with the sequent $\Delta \vdash |k| \leq \mathbf{M}$ open, i.e., there is currently no justification for Δ entails $|k| \leq \mathbf{M}$. The line

L3. $\quad \emptyset \quad \vdash b = k * a\sigma + l \quad$ (`CAS`;L4)

(from the `ComplexEstimate` method below) means that the computer algebra system CAS is expected to justify the formula $b = k * a\sigma + l$ in line L3 with the help of the formula in line L4. Similarly an automated theorem prover or a decision procedure could be specified here.

3.2 Meta-Level Reasoning

Although most methods encapsulate a chunk of calculus-level proofs, and therefore, proof plans are generally much shorter than the corresponding expansions into a calculus-level proof, the potential search spaces are still prohibitively large. The reason is that a large number of alternative methods is usually applicable at each choice point and also, more seriously, the search space for mathematical proofs is potentially infinite. That is to say, even for a finite number of operators, there may be infinitely many branches at each choice point, for example, when existentially quantified variables have to be instantiated or when a lemma has to be introduced.[7] Hence, special techniques for reducing and guiding the search are needed in proof planning.

Informed search is superior in domains, where control knowledge exists and mathematics is surely a field where such knowledge has been accumulated. The good news is that, since methods represent mathematically meaningful steps, control knowledge can express mathematical 'intuition' rather than just syntactical information such as an order or a number restriction on literals as in traditional ATP. The bad news is that it can be hard to extract this knowledge and to represent it appropriately.

Our representation of control knowledge depends on the kind of the knowledge, in particular on whether it expresses legal or heuristic, local or global knowledge. As opposed to 'local' knowledge as defined above, 'global' means that its evaluation procedure may have to inspect the whole PDS and its history (i.e., the failed proof attempts etc.) as well as time resources, the user model, and other global settings such as the theory within which the problem is stated and the typical examples of this theory. While legal and local knowledge is appropriately encoded into the *application conditions* of methods, heuristic knowledge should be encoded into control-rules. They can then be used for meta-level reasoning which is known to be extremely important in mathematics [109].

We prefer an explicit and modular representation of heuristic control knowledge in declarative control-rules rather than the previous representation hardwired into the planner or the methods. This is useful because the same method can be used in different theory contexts with different control. Consider, e.g., a method that translates a goal into a statement about natural numbers, i.e., into a goal that can then be shown by Peano Induction. The function that determines the natural number is a parameter of a very general 'translation' method. In many completeness proofs for logic calculi this natural number is either the excess literal number [3], the length of clauses, or the number of literal occurrences. In other theories the choice of this number may be different, e.g., the length of a derivation, the complexity of a formula, the number of grammar rule applications etc. Now, the method that finds this natural number is very general and applicable in many mathematical domains, whereas the control knowledge that determines the possible range of the parameter (the function) is specific and thus usually different in different domains.

[7]This is a problem well known, e.g., in induction and verification. The ultimate reason for infinite branching is that the cut rule, i.e., the rule

$$\frac{\Gamma, B \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A},$$

is applied backward and this cannot generally be avoided in mathematics. For an analysis see [68].

Other well known reasons for using control-rules are: Modularly implemented and declaratively represented control-rules ease the *modification and extension* of the control knowledge. For instance, when new control information is required for new cases of problem solving, the existing control-rules can be modified or new control-rules can be introduced easily, as opposed to a re-implementation of the procedurally implemented control component of a planner. In particular, when the control is changed or *generalized* or when new methods are introduced into the domain, no re-implementation of the affected methods is necessary. Moreover, the declarative representation of control information by rules can be a basis for *automatically learning* control-rules, as realized in some planning systems, e.g., in [96, 19, 71].

In ΩMEGA, a control unit evaluates the control-rules and guides the proof planner in its choice between several candidates, similar to an expert system, where the list of candidates rules is reduced to a smaller list. Control-rules have an antecedent and a consequent. The antecedent specifies the overall context within which this control-rule is applicable; its evaluation determines whether the control-rule is rejected or not. Currently, the consequent of a rule encodes the advice to *select* a specified subset of the available candidates, to *reject*, or to *prefer* one candidate over another candidate. The first two types of rules prune the search space, while prefer-rules change the default order without excluding other alternatives.

Corresponding to the type of the choices of the planner we have the following classes of control-rules in ΩMEGA:

- `method-choice` for choosing among several methods, this may come with binding choices,

- `sequent-choice` for choosing among goals (in backward planning) or among assumptions (in forward planning),

- `strategy-choice` for choosing a refinement strategy (not discussed in this article, but see [89]).

The rules are formulated in an expressive meta-language. The decidable meta-predicates in the antecedents of control-rules inspect the PDS and the planning state (e.g., `goal-matches`), the planning history (e.g., `last-method`), the constraint state (e.g., `unique-value`), the available resources, the user model, the theory in which to plan, including typical models of the theory(e.g., `invalid-in-typical`). For example, the following rule `case-analysis-intro` which is a reconstruction of a critic in *CL^AM* expresses the heuristic that if the method `Rewrite` whose parameter is instantiated by a rule `(C -> R)` is not applicable because the formula `C` is not trivially provable, then a `CaseSplit` method should be introduced into the plan.

```
(control-rule case-analysis-intro
        (kind method-choice)
        (IF  (last-method (Rewrite (?C -> ?R))) AND
                  (failure-condition (trivial ?C)))
        (THEN (select (CaseSplit (?C or not ?C)))))
```

For instance, if `Rewrite`(ϕ) was tried in the last planning step with the instantiation of ϕ: $\equiv [x \neq a \rightarrow \frac{f(x)-f(a)}{x-a} = f'(a)]$ but this failed because $x \neq a$ does not hold, then a `CaseSplit` on $(x \neq a \vee x = a)$ is chosen next in order to enable `Rewrite`(Φ) in one of the case-split branches.

The use of control-rules gives rise to an extended planning algorithm as shown in Table 2 below (see Table1 for comparison).

`Backwards-Refine-PlanC`(π, G)

Termination: **if** goal agenda G empty, **then** `Solution`.
if no operator applicable, **then** `Fail`.

Goal Selection: Evaluating control-rules (for goals) returns G' with $G' \subseteq G$.
Choose open goal $g \in G'$.

Operator Selection:

- Evaluating control-rules (for operators) returns $O' \subseteq O$.
- For each operator $op \in O'$

  for each $\oplus$-effect e of op

  let σ be the matcher $e\sigma = g$

  if application-conditions($op\ \sigma$) $= true$,

  then op is applicable.

- Choose one applicable M $\in O'$ (backtracking point)
  - insert M into π
  - insert constraints into π
  - update G.

Recursion: Call `Backwards-Refine-PlanC` on the refined partial plan π.

Table 2: Outline for controlled backward planning

3.3 Integrating External Reasoning Systems

In many mathematical proofs, logical steps are naturally combined with some form of specialized reasoning or effective computation such as computing integrals, solving equations, or solving inequalities (see [25] for an overview). This specialized reasoning does not work too well with traditional ATP systems whereas theory-specific systems and algorithms can perform such services more efficiently because they represent objects such as rational and real numbers [52] by specialized data types that can be efficiently handled and because they rely on efficient special-purpose algorithms called background reasoning in [25].

Proof planning provides a natural framework for integrating such external reasoning systems as follows. Suppose we have an algorithm for the computation of the greatest common divisor, GCD whose computation implicitly uses axioms and theorems for integers. This computation can be first included into the proof plan by wrapping the call of the GCD-algorithm into a method whose tactic computes the output of the method. In case this computation cannot be trusted,[8] or in either case we prefer a sequence of calculus-level steps for the proof anyway, the computation has to be expanded into a proof that is a subproof of the overall calculus-level proof. Therefore, in ΩMEGA this expansion is possible in principle. However, generating a proof plan from a system's trace is expensive and hence the plan generation mode is called on demand only.

In the following, we shall briefly mention the external reasoning systems integrated into ΩMEGA and then concentrate just on those aspects that are relevant for this article.

3.3.1 Traditional Automated Theorem Provers and Decision Procedures

The first systems that were integrated into ΩMEGA are the MKRP system [36] and OTTER [81]. Currently, more provers are integrated, e.g., SPASS [27], PROTEIN [10], Bliksem [99], EQP [79], and Waldmeister [49]. The ultimate goal of this integration is this: once we have an abstract proof plan there may be many gaps in the overall proof that could easily be solved by a call of one of these ATP-systems. The nontrivial aspects of this integration is to spot the right gaps and secondly to translate the resulting proof into a sequence of ND-rules which is the chosen base logic of ΩMEGA. The second problem is solved in ΩMEGA [55] and other systems [4, 101, 95, 75] while typically the first problem is currently circumvented by interactively calling the ATPs.

3.3.2 Computer Algebra Systems

In order to execute symbolic computations more efficiently, ΩMEGA integrates several computer algebra systems including $\mu\mathcal{CAS}$ [64], an experimental CAS, to simplify algebraic expressions and to compute terms in the proof planning process.

The call of a computer algebra system is wrapped into a method or in a function that is invoked when evaluating a method's *application conditions*. The computer algebra system returns a simplified expression or a newly computed term. A node in the proof plan that is justified by `CAS` can be expanded recursively into an ND-proof that can be proof checked. For this expansion, $\mu\mathcal{CAS}$ runs in a plan-generation–mode which returns the protocol information from which a proof plan justifying its computation can be re-constructed. This plan can then be recursively expanded into an ND-proof (see [64]).

3.3.3 Constraint Solvers

Many proofs require the construction of an object with theory-specific properties. This is usually indicated by an existentially quantified variable in the problem. In traditional theorem proving this construction is carried out by the unification algorithm.

[8] E.g., a computation may not check whether the divisor is zero.

Our solution to this problem is to delay this instantiation as much as possible and to integrate an external constraint solver that incrementally restricts the possible object values, essentially as in constraint logic programming (CLP) [59]. This process is essential for the purpose of this paper and, as we shall see, the use of constraint solvers differs from that of a CAS or an ATP as it requires the permanent presence during the planning process and the collection and propagation of many constraints by the constraint solver (see [91]).

Our external constraint solver uses the notation and functionalities common to CLP. These are outlined next followed by a description of a generic interface with proof planning. The common theory of CLP [60] defines a constraint domain ($\mathcal{D}$,$\mathcal{L}$) for a given signature Σ that includes the symbol $=$. $\mathcal{D}$ is a Σ-structure, i.e. an interpretation for Σ which interprets $=$ as the identity, and $\mathcal{L}$ is a class of first-order Σ-formulae (constraints) that is closed under variable renaming, conjunction, and existential quantification. For instance, the Σ-structure $\mathcal{R}$ is the constraint domain of arithmetic over the real numbers for $\Sigma = (0, 1, +, *, =, <, \leq)$, where $+, *$ are interpreted as the usual addition and multiplication function and $<, \leq$ are interpreted as the less-than and less-than-or-equal relations on reals, respectively.

Basic constraints are those for which satisfiability can be decided directly, e.g., $(X < a)$. *Non-basic constraints* have only incomplete decisions, e.g., $X + Y = Z$ when only one variable is known. *Conditional constraints* have the form (if c then A_1 else A_2), for a constraint c [114].

Jaffar and Maher [60] introduce an operational semantics for CLP systems as state transition systems. In order to implement the abstract operational model, the following functions have to be realized:

- Initialization of the constraint state,
- Consistency check,
- Entailment check,
- Propagation of constraints, including simplification,
- Reflection of the constraint state
- Search for the instantiation of variables.

The interface functions `tell` and `ask` can pass constraints to the solver. The operation `tell`(c) passes a constraint c to the constraint solver and then propagates the constraint in case c is consistent with the constraint store. In this case, it returns *true* and the new constraint state. Otherwise it returns *fail*. The operation `ask` passes a conditional constraint (if c then A_1 else A_2) to the constraint solver for testing entailment of c from the constraint store. It returns A_1, if c is entailed and A_2 otherwise. For instance, `ask`(if $0 \leq x$ then `tell`$|x| = x$) checks whether $(0 \leq x)$ is entailed by the constraint store, and if so, then $(|x| = x)$ is `told` to the constraint solver.

The integration of a constraint solver into proof planning serves several purposes: *First*, it is used in the process of proof planning to determine whether a certain method can be legally applied and *secondly*, the constraint state can be reflected onto an *answer constraint* C, and

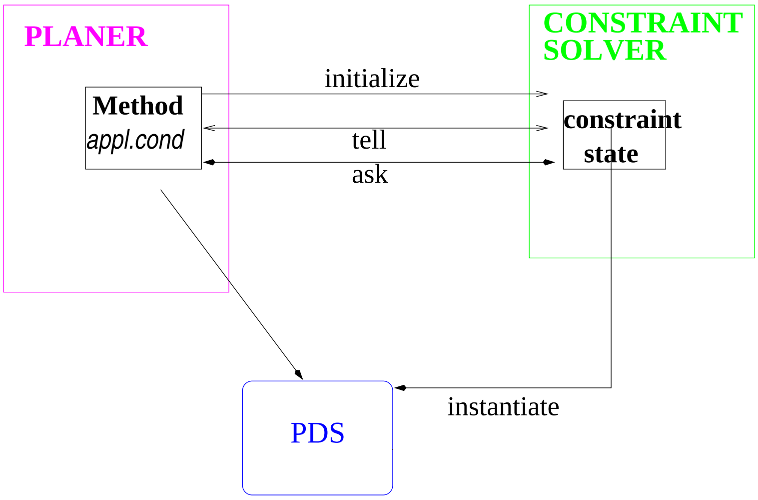

Figure 4: Integration of a constraint solver into proof planning

thirdly the constraint solver searches for instantiations of implicitly existentially quantified variables.

Figure 4 summarizes the integration of a constraint solver into proof planning. Some methods that are available to the planner are interfaced with the constraint solver by the functions `tell` or `ask` that are called when the *application conditions* are evaluated: the constraint solver may compute a reflection C of the constraint state. When the plan is completed, an `instantiation` function instantiates the meta-variable $\mathcal{C}$ by the formula C at each occurrence in the PDS [87].

A method interfacing proof planning with the constraint solver `CS` is `Solve-b` that is shown below. The `b` in the method's name indicate that it is employed in backward planning and `CS` is the parameter determining the particular constraint solver. `CS` stands for any constraint solver and can be instantiated by, e.g., a constraint solver for linear arithmetic in the real numbers, for finite domains, or for set theory.

method: `Solve-b(`c`,CS)`	
premises	L1
conclusions	$\ominus$L2
appl.cond	constraint(c,`CS`) **AND** **IF** var-in(c) **THEN** `tell`(c) **ELSE** `ask`(if c then *true* else *fail*)
proof schema	L1. $\mathcal{C}$ $\vdash \mathcal{C}$ (HYP) L2. $\Delta, \mathcal{C}$ $\vdash c$ (solvCS;L1)

c is the constraint goal that is closed by `Solve-b`. The *proof schema* of `Solve-b` contains a meta-variable $\mathcal{C}$ for the answer constraint of `CS`. The instantiation of $\mathcal{C}$ is relevant for line L2 in the *proof schema* that suggests that the constraint can logically be derived from the (yet unknown) answer constraint. Hence, the value for $\mathcal{C}$ is used in the recursive expansion of

`Solve` steps. The *proof schema* of `Solve-b` contains a line with the line-justification solvCS which names the method that derives c from the instantiation of the meta-variable $\mathcal{C}$ by eliminating conjunctions repeatedly.

The *application conditions* of the `Solve-b` method determine whether a constraint goal is handled by `tell` or by `ask`. The access of the constraint solver via `tell` is chosen when the constraint at hand contains an implicitly existentially quantified variable. Otherwise, `ask` is chosen. The reason is that new constraint goals that contain only constants and universally quantified variables cannot be introduced into the constraint store without loss of generality, whereas constraints with implicitly existentially quantified variables can.[9] The *application conditions* are satisfied if `ask` returns *true*, i.e., c is entailed by the current constraint store, or if `tell` returns *true*, i.e. the constraint store is consistent with c. This gives an account on how the constraint solver is involved in checking the legal applicability of a `Solve` method.

The second task of the constraint solver, reflecting the constraint state onto an answer constraint, is currently performed with the final constraint state. The resulting answer constraint formula C is an assertion about the value restrictions of the implicitly existentially quantified variables.

Why is it necessary to introduce $\mathcal{C}$ anyway? Suppose, we collect restrictions by backward planning and these are condensed in an answer constraint formula C. In a formal proof consisting of forward inference steps, C is stated at the beginning and subsequent proof steps have to refer to C as an hypothesis, i.e., the hypothesis must occur in goals before it is actually known. Hence, in order to obtain a correct ND-level proof after the recursive expansion, a meta-variable $\mathcal{C}$ is introduced as a place holder for the formula C. This is realized in backward planning by the `InitializeCS(`Th`)` method that reduces a goal

$$\Delta \vdash thm$$

to the subgoals

$$\Delta, \mathcal{C} \vdash thm, \qquad \Delta, Th \vdash \mathcal{C},$$

where Th is the theory of the constraint solver and where $\mathcal{C}$ holds the place for a formula C.

`InitializeCS` combines the ND-rules $\rightarrow$I and $\rightarrow$E.[10] It contributes to the hierarchization of proof planning.

method: `InitializeCS(`T`)`				
premises	$\oplus$ L1			
conclusions	$\ominus$ L4			
appl.cond				
proof schema	L1.	$\Delta, \mathcal{C}$	$\vdash thm$	(OPEN)
	L2.	Δ	$\vdash \mathcal{C} \rightarrow thm$	($\rightarrow$I;L1)
	L3.	Δ, T	$\vdash \mathcal{C}$	(OPEN)
	L4.	Δ, T	$\vdash thm$	($\rightarrow$E;L2,L3)

[9] The introduction of a constraint assumption into the constraint store by forward planning works differently because proof assumptions can always be introduced.

[10] $\rightarrow$I(ntroduction) is $\frac{\Delta, F \vdash G}{\Delta \vdash F \rightarrow G}$. $\rightarrow$E(limination) is $\frac{\Delta \vdash F \rightarrow G, \Delta \vdash F}{\Delta \vdash G}$.

Note that – as a form of precondition abstraction – L3 is visible as a subgoal only after the method's expansion because it does not occur in the *premises*. Only when the instantiation of $\mathcal{C}$, C, can also be proven, is the proof completed.

The parameter T in `InitializeCS` stands for a theory T for which a particular constraint solver CS_T is employed, e.g., set theory or linear arithmetic in $\mathcal{R}$.

3.4 Extension of (Mathematical) Theories

In knowledge-based proof planning the mathematical domain knowledge consists not only of axioms, definitions, and theorems but also of methods, control-rules, and domain-specific external reasoners. This knowledge is hierarchically organized and stored in *theories*. ΩMEGA's general-purpose proof planner can access a theory base, called MBase [39], that contains domains such as *Base*, *Set Theory*, *Calculus* and that is currently extended to further mathematical theories as well.

4 A Case Study: Proof Planning of Limit Theorems

We shall now demonstrate knowledge-based proof planning in operation, using the limit theorems as our domain. These theorems are formulated and proved in the theory $\mathbb{R}$ of the real numbers.

In the remainder, $/, *, +, -, ||$ denote the division, multiplication, addition, subtraction, and the absolute value function in $\mathbb{R}$, respectively. Prolog notation is used for constants and variables.

Limit theorems claim something about the limit $\lim_{x \to a} f(x)$ for a function f or about continuity. Since the formal definition of $\lim_{x \to a} f(x)$ is

$$\forall \epsilon (0 < \epsilon \to \exists \delta (0 < \delta \wedge \forall x (x \neq a \wedge |x - a| < \delta \wedge x \neq a \to |f(x) - l| < \epsilon))), \qquad (2)$$

the standard proofs of these theorems are often called ϵ–δ–proofs, i.e., proofs that postulate the existence of a δ such that a conjecture of the form $\ldots |X| < \epsilon$ can be proven under assumptions of the form $\ldots |Y| < \delta$. The class of limit theorems includes the theorem LIM+ that states that the limit of the sum of two functions is the sum of their limits. The following sequent formalizes LIM+[11]

$$\lim_{x \to a} f(x) = l_1 \wedge \lim_{x \to a} g(x) = l_2 \vdash \lim_{x \to a} (f(x) + g(x)) = l_1 + l_2, \qquad (3)$$

and after expanding, the definition of $\lim_{x \to a}$, becomes

$$\begin{array}{l}
\forall \epsilon_1 (0 < \epsilon_1 \to \exists \delta_1 (0 < \delta_1 \wedge \forall x_1 (x_1 \neq a \wedge |x_1 - a| < \delta_1 \to |f(x_1) - l_1| < \epsilon_1))) \wedge \\
\forall \epsilon_2 (0 < \epsilon_2 \to \exists \delta_2 (0 < \delta_2 \wedge \forall x_2 (x_2 \neq a \wedge |x_2 - a| < \delta_2 \to |g(x_2) - l_2| < \epsilon_2))) \\
\vdash \forall \epsilon (0 < \epsilon \to \exists \delta (0 < \delta \wedge \forall x (x \neq a \wedge |x - a| < \delta \to |(f(x) + g(x)) - (l_1 + l_2)| < \epsilon)) \ .
\end{array}$$

[11] Alternatively LIM+ can be formalized by the assumptions $\lim_{x \to a} f(x) = l_1$ and $\lim_{x \to a} g(x) = l_2$ and the goal $\lim_{x \to a} (f(x) + g(x)) = l_1 + l_2$

Similar theorems in this class are LIM- and LIM* for the difference and product of limits; the theorem ContinuousComp states that the composition of two continuous functions is continuous;[12] Continuous+ states that the sum of two continuous functions is continuous and similarly Continuous* and Continuous-, and finally there is a myriad of theorems about the limits of polynomial functions like $\lim_{x \to a} x^2 = a^2$.

In 1990, Woody Bledsoe [17] proposed several versions of LIM+ as a challenge problem for automated theorem proving. The simplest versions of LIM+ (problem 1 and 2 in [17]) are at the edge of what traditional automated theorem provers can prove today (see the comparison in § 4.4) but, certainly, LIM* is well beyond their capabilities.

One reason why these proofs are difficult for a system is due to the alternating quantifiers which require the construction of a δ dependent on a variable ϵ such that certain estimations hold. This is a non-trivial thing to do and difficult for a student as reported, e.g., in [72].

In most textbooks, the intelligent instantiation of δ comes out of the blue which is, of course, puzzling for a freshman. The typical way a mathematician goes about to discover the proof of such a theorem is to (incrementally) restrict the possible values of δ as e.g., recorded in the textbook of Bartle and Sherbert "Introduction to Real Analysis" [9]. The two authors give a recipe on how to proceed, namely an incremental restriction of a natural number k when proving a theorem about the products of limits of sequences (x_n) and (y_n), where the definition of $\lim_{n \to \infty}(x_n) = x$ is:

$$\forall \epsilon (0 < \epsilon \to \exists k (0 < k \wedge \forall n (n > k \to |x_n - x| < \epsilon))). \tag{4}$$

The natural number k in this definition (4) corresponds to the real number δ in the definition (2) of limits of functions. In the proof for products of the limits x and y of the sequences (x_n) and (y_n) respectively, Bartle and Sherbert introduce the auxiliary variables M_1 and M upon which K depends: "According to Theorem ... there exists a real number $M_1 > 0$ such that $|x_n| \leq M_1$ for all $n \in \mathbb{N}$ and we set $M = sup\{M_1, |y|\}$. Hence, we have the estimate

$$|x_n * y_n - x * y| \leq M * |y_n - y| + M * |x_n - x|.$$

From the convergence of (x_n) and (y_n) we conclude that if $\epsilon > 0$ is given, then there exist natural numbers k_1 and k_2 such that if $k_1 \leq n$, then $|x_n - x| < \epsilon/2M$, and if $k_2 \leq n$, then $|y_n - y| < \epsilon/2M$. Now let $k(\epsilon) = \sup\{k_1, k_2\}$, then if $k(\epsilon) \leq n$ we infer that

$$|x_n * y_n - x * y| \leq M * (\epsilon/2 * M) + M * (\epsilon/2 * M) = \epsilon.$$

Since ϵ is arbitrary, this proves that the sequence $X * Y$ converges to $x * y$." [9]

Inspired by a similar mathematical idea, Bledsoe, Boyer, and Henneman implemented the limit heuristic for their special-purpose theorem prover, IMPLY [18] which proves formulae of the form $|A| < \epsilon_1 \to |B| < \epsilon$ by representing B as a linear combination $B = k * A + l$, and by proving the simpler formulae $|k| < M$, $|A| < \epsilon/2 * M$, and $|l| < \epsilon/2$, containing a new variable M. We shall reconstruct this trick in one of our methods, $\texttt{ComplexEstimate}_<$, below.

[12] The definition of the relation *continuous(f)* at a point a builds on $\lim_{x \to a} f(x)$.

4.1 Methods for ϵ-δ-Proofs

In proving limit theorems, frequently the magnitude of a term has to be estimated, that is, we need estimation methods. One of them is $\texttt{ComplexEstimate}_<$, a method for estimating the magnitude of the absolute value of complex terms such as $(f(x)+g(x))-(l_1+l_2)$. The following is the frame representation of this method.

method: $\texttt{ComplexEstimate}_<(a,b,e_1,\epsilon)$	
premises	L1, $\oplus$L2,$\oplus$L3, $\oplus$L4
conclusions	$\ominus$ L17
appl.cond	$\exists\sigma. GetSubst(a,b)=\sigma$ **AND** $\exists k,l. CASsplit(a\sigma,b)=(k,l)$
proof schema	L1. Δ $\vdash$ $\|a\|<e_1$ () L2. Δ $\vdash$ $\|k\|\leq \mathbf{M}$ (OPEN) L3. $\vdash$ $\|a\sigma\|<\epsilon/(2*\mathbf{M})$ (OPEN) L4. Δ $\vdash$ $\|l\|<\epsilon/2$ (OPEN) L5. $\vdash$ $b=b$ (Ax) L6. $\vdash$ $b=k*a\sigma+l$ (CAS;L5) L7. $\vdash$ $0<M$ (OPEN) L17. Δ $\vdash$ $\|b\|<\epsilon$ (fix;L6,L7,L1,L2, L3,L4)

This method reduces a goal that is a difficult estimation by three simpler estimation goals in case there is an assumption matching the formula of line L1 in the planning state. Each $\texttt{ComplexEstimate}_<$ step reduces a goal matching the formula of line L17 to subgoals that are instances of the formulae in lines L2, L3, and L4, respectively.

Each application of $\texttt{ComplexEstimate}_<$ suggests the existence of a real number $\mathbf{M}$ whose value is restricted by the inequalities in line L2 and L3. This $\mathbf{M}$ is then used to propagate the given value restrictions from the implicitely universally quantified variables to implicitly existentially quantified variables as described in section 4.3 below.

The *application condition* evaluates to *true* if there is a substitution σ and terms k and l such that b can be represented as a linear combination of $a\sigma$, $b=k*a\sigma+l$, where a is the term of line (0) that is in the current planning state. In this case, $\texttt{ComplexEstimate}_<$ is applicable.

The *proof schema* contains a schematic proof of $\Delta\vdash|b|<\epsilon$ from $b=k*a\sigma+l$ and $0<$ M, and from the formulae in lines L1, L2, L3, and L4, respectively. The line-justification in L1 CAS denotes the application of a CAS that has to verify the equation $b=k*a\sigma+l$. Note that the line L7 does not occur in the $\oplus$-premises although its justification is OPEN, i.e., it does not have an actual justification yet and, thus, its satisfaction is postponed. In the line-justification of L12, 'fix' abbreviates a whole subproof that employs, among other things, the Triangle Inequality $|X+Y|\leq|X|+|Y|$. An explicit *proof schema* that details 'fix' looks as follows.

Line	Hyps		Formula	Reason
L1.	Δ	$\vdash$	$\|a\|<e_1$	(assumption)

L2.	Δ	$\vdash$	$\|k\| \leq \mathrm{M}$	(OPEN)
L3.	Δ	$\vdash$	$\|a\sigma\| < \epsilon/(2 * \mathrm{M})$	(OPEN L1)
L4.	Δ	$\vdash$	$\|l\| < \epsilon/2$	(OPEN)
L5.		$\vdash$	$b = b$	(Ax)
L6.		$\vdash$	$b = k * a\sigma + l$	(`CAS`)
L7.		$\vdash$	$0 < \mathrm{M}$	(OPEN)
L8.		$\vdash$	$\|b\| \leq \|k * a\sigma\| + \|l\|$	(triang;L6)
L9.		$\vdash$	$\|b\| \leq \|k\| * \|a\sigma\| + \|l\|$	(Mval;L8)
L10.	Δ	$\vdash$	$\|k\| * \|a\sigma\| + \|l\| \leq M * \|a\sigma\| + \|l\|$	(mult$\leq$;L2)
L11.	Δ	$\vdash$	$\|b\| \leq \mathrm{M} * \|a\sigma\| + \|l\|$	(trans$\leq$;L9,L10)
L12.	Δ	$\vdash$	$\mathrm{M} * \|a\sigma\| < M * \epsilon/(2 * \mathrm{M})$	(mult$<$;L3)
L13.	Δ	$\vdash$	$\mathrm{M} * \|a\sigma\| + \|l\| < M * \epsilon/(2 * \mathrm{M}) + \|l\|$	(add$<$;L12)
L14.	Δ	$\vdash$	$\|b\| < \mathrm{M} * \epsilon/(2 * \mathrm{M}) + \|l\|$	(trans$<$;L11,L13)
L15.	Δ	$\vdash$	$\mathrm{M} * \epsilon/(2 * \mathrm{M}) + \|l\| < M * \epsilon/(2 * \mathrm{M}) + \epsilon/2$	(add$<$;L4)
L16.	Δ	$\vdash$	$\|b\| < \mathrm{M} * \epsilon/(2 * \mathrm{M}) + \epsilon/2$	(trans$<$;L14,L15)
L17.	Δ	$\vdash$	$\|b\| < \epsilon$	(simpl;L16)

where 'triang' means an application of the Triangle Inequality, 'trans' means an application of the transitivity theorem, etc.
The planner handles the method `ComplexEstimate`$_<$ as follows.

- If an open goal matches the formula of L17 and if an assumption that matches the formula in line L1 is in the planning state, then the parameters a, b,e_1, ϵ of `ComplexEstimate`$_<$ are instantiated by this matcher.

- Now the *application conditions* are evaluated, that is, first the function $GetSubst(a, b)$ is invoked which returns, if successful, a substitution σ. In case *GetSubst* has been successfully executed, the function *CASsplit* calls a computer algebra system with the arguments $a\sigma$ and b and may return two terms k and l such that $b = k * a\sigma + l$. If successful, the result instantiates the variables k and l and the *application conditions* evaluate to *true*. Then the method is applicable

- When applicable, the planner inserts `ComplexEstimate`$_<$ into the PDS, the goal L17 is deleted from the planning state, and the new goals corresponding to lines L2, L3, and L4 are added to the planning state.

- When later on the method is expanded, the *proof schema* gets inserted into the PDS, and now this expansion postulates the formula $0 < \mathbf{M}$ as a new open subgoal. That is, line L7 becomes an open subgoal in the next lower planning level which is one way to benefit from a *hierarchical* planning process. Further recursive expansion of line L6 with the justification CAS, will call the computer algebra system $\mu\mathcal{CAS}$ which runs in plan-generation mode and returns a proof plan for the justification of line L6.

In planning LIM+, at some point the goal $\Delta \vdash |f(x) + g(x) - (l_1 + l_2)| < \epsilon$ is to be proven when the assumption $\Delta \vdash |f(X_1) - l_1| < E_1$ is available. In this sequent, Δ is a set of assumptions and X_1, E_1 are implicitly existentially quantified variables. When the *application conditions* are evaluated *GetSubst* returns the substitution $[x/X_1]$ and *CASsplit* returns the list $(1, (g(x) - l_2))$ since the parameter a is instantiated by $(f(X_1) - l_1)$ and the parameter b is instantiated by $(f(x) + g(x) - (l_1 + l_2))$. Now the goal $\Delta \vdash |f(x) + g(x) - (l_1 + l_2)| < \epsilon$ is replaced by the new goals

(1) $\Delta \vdash |1| \leq \mathbf{M}$,

(2) $\Delta \vdash |f(X_1) - l_1| < \epsilon/(2{*}\mathbf{M})$,

(3) $\Delta \vdash |g(x) - l_2| < \epsilon/2$.

The attentive reader will have noticed that the numerous axioms and the theorems of the domain theory, such as the Triangle Inequality and others, are conspicuously absent; they are invisible at the planning level ! Only when a method such as `ComplexEstimate`$_<$ is expanded and the *proof schema* is inserted into the proof plan, are the appropriate axioms imported from a theory. This is a very natural way to prove a theorem and it is a means to avoid the common paradoxical situation of traditional automated theorem proving, where exactly those axioms, definitions, and theorems that are needed in the particular proof have to be stated beforehand, i.e., before we even know the proof. It also avoids many other problems that plague traditional systems: the active axioms are kept at a minimum and the problem that certain axioms such as commutativity or associativity seduce the system into senseless behaviour (called semantic noise or trashing [6, 110]) as we use the axiom now in a goal-directed way only for a special well defined purpose.

While `ComplexEstimate`$_<$ proves (in)equalities by decomposition, other methods such as `Solve-b`, `Solve-f` (see section 4.3), and `Solve*` treat (in)equalities more directly. `Solve`* is a method that first reduces a goal $t_1 < t_2$ with the help of an assumption $t_1' < t_2'$, where t_1 and t_1' are unifiable by a substitution σ, to a subgoal $t_2\sigma' < t_2\sigma$ and then tries to remove $t_2\sigma' < t_2\sigma$ by a `Solve-b` method.

`UnwrapHyp`, a method used in forward planning, highlights a subformula of an assumption by the `Focus` method, and then it applies various other methods (e.g., AndElimination, ImpliesElimination) in order to extract the highlighted subformula as a single assumption. In other words, this method 'unwraps' this subformula out of the original assumption. The ultimate goal of this method in proving limit theorems is to prepare an assumption such that it can be used as the L1 assumption in `ComplexEstimate`$_<$.

The method `RemoveFocus` removes a focus which was set by `Focus` beforehand, the method `Normalize` specifies a tactic that calls submethods such as `ImpliesIntroduction` and `AndIntroduction`. All in all, we have less than two dozen methods for this (small) mathematical area that by and large correspond to what a student would have to learn in order to master the field.

4.2 Control-Rules

Even the most appropriate methods do not imply that a plan can be found automatically. In fact, our planner cannot find a plan for LIM+ without additional control knowledge. For this reason, we manually extracted mathematical problem solving behaviour typically used for proving limit theorems and translated it into control knowledge explicitely represented by control-rules. We demonstrate this for the following very simple problem solving behaviour. (More complicated meta-reasoning is necessary, e.g., for finding proofs by contradiction. This is, however, beyond the scope of this paper.)

- Linear inequalities can be shown by a direct estimation or by a decomposition of the term to be estimated. For the latter, often a proof assumption has to be refered to.

This can be translated into the following – verbally expressed – control knowledge for proof planning,

- Linear inequality goals can be removed by one of the methods `Solve-b`, `Solve*`, or by `ComplexEstimate`$_<$. The latter requires some preparation by `UnwrapHyp` which extracts a subformula s from an assumption. Afterwards `ComplexEstimate`$_<$ can use the assumption s.

In turn, this knowledge can be formally encoded into the control-rule

- ```
  (control-rule prove-inequalities
        (kind method-choice)
          (IF (goal-matches (?goal (?x < ?y))))
          (THEN (prefer ((Solve-b ?goal)
                         (Solve* ?goal)
                         (ComplexEstimate< ?goal)
                         (UnwrapHyp ?goal)))))
  ```

This rule together with the following control-rules are sufficient to successfully plan all the above mentioned limit theorems.

```
(control-rule CS-Introduction
      (kind method-choice)
        (IF (last-method Skolemize))
        (THEN
          (prefer (InitializeCS))))

(control-rule Solve-f-first
      (kind method-choice)
        (IF (inequality-assumption ?assumption))
        (THEN
          (prefer (Solve-f ?assumption))))
```

The latter ensures that any (in)equality assumption is passed to the constraint solver as soon as possible.

4.3 Computer Algebra and Constraint Solving

As described above, a *computer algebra system* computes instantiations of certain terms when the *application conditions* of `ComplexEstimate`$_<$ (*CASsplit*) are evaluated. Any such instantiation has to be verified later on, when the node justified by `CAS` is expanded into a checkable ND-proof.
```

Our *propagation-based constraint solver* COSIE works for the constraint domain $\mathcal{R}$ which has been extended by the interpreted absolute value function $\lambda x.|x|$ and the division function $\lambda x \lambda y. x/y$. Constraints are (in)equalities, i.e., formulae of the form $x < y$, $x \leq y$, or $x = y$. The constraint store is represented by sets of intervals described by sets of lower and upper bounds. A new constraint is introduced into the store, simplified, and propagated, if it is consistent with the current constraint store.

The purpose of the methods `Solve-b` (`Solve-f`) is to remove a simple (in)equality goal (to employ a constraint assumption (by adding it to the constraint store or by checking entailment. See section 3).

`Solve-b` is a candidate method as soon as there is an (in)equality goal. It is applicable if no implicitly existentially quantified variable occurs in the goal and the goal is entailed by the current constraint store or if an implicitly existentially quantified variable occurs in the goal, and if this constraint goal is consistent with the constraint store. The latter is checked by `tell`. All of this information is represented in *appl.cond.*

For instance, while planning LIM+, `Solve-b`$(0 < \delta_1)$ is applicable because no implicitly existentially quantified variable occurs in this goal and $(0 < \delta_1)$ is entailed by the constraints $(0 < D)$ and $(D \leq \delta_1)$ from the constraint store. Furthermore, `Solve-b` is applicable to the subgoal $(|1| \leq \mathbf{M})$ because the implicitly existentially quantified variable $\mathbf{M}$ occurs in the goal and $(|1| \leq \mathbf{M})$ is consistent with the constraint store.

The method `Solve-f` handles constraint assumptions in forward planning. For example, while planning LIM+, `Solve-f` introduced $(0 < D)$ into the constraint store as a constraint hypothesis.

To summarize, while planning the proof of the LIM+ theorem, the `Solve` steps (`Solve-b` and `Solve-f`) `tell` the following sequence of constraints to the constraint solver

$$0 < \epsilon,\ 0 < D,\ |1| \leq \mathbf{M},\ 0 < \mathbf{M}\ E_1 \leq \epsilon/(2 * \mathbf{M}),\ x = X_1,\ E_2 \leq \epsilon/2,\ x = X_2\ D \leq \delta_2,$$
$$D \leq \delta_1,\ 0 < E_1,\ 0 < E_2,\ 0 < \delta_1,\ 0 < \delta_2,$$

where the variables D, E_1, E_2 and the (Eigen)variables $\delta_1, \delta_2, \epsilon$ are those of the original planning problem and $\mathbf{M}$ is the auxiliary variable introduced by `ComplexEstimate`$_<$. Now the upper bound $\epsilon/2$ for E_1 is propagated from the constraints $1 \leq \mathbf{M}$ and $E_1 \leq \epsilon/(2*\mathbf{M})$. This leads to the following final constraint store

$$\begin{array}{ccccc}
0 & < & \delta_1 & < & +\infty; \\
0 & < & \delta_2 & < & +\infty; \\
0 & < & \epsilon & < & +\infty; \\
0 & < & E_2 & \leq & \epsilon/2; \\
0 & < & D & \leq & \delta_2, \delta_1; \\
0 & < & E_1 & \leq & \epsilon/(2 * \mathbf{M}), \epsilon/2; \\
1 & \leq & \mathbf{M} & \leq & \epsilon/(2 * E_1); \\
-\infty & < & X_1 = x = X_2 & < & +\infty
\end{array}$$

In other words, we have that a lower bound for E_2 is 0 and an upper bound for E_2 is $\epsilon/2$; a lower bound for D is 0 and the upper bounds are δ_1, δ_2, etc, and the attentive reader familiar with ϵ-δ-proofs will no doubt have noticed that this is exactly the sequence of events a good student would have to go through in a maths class.

At the end of the session, a reflection of the final constraint store removes redundant bounds such as $\epsilon/(2 * \mathbf{M})$. The reflection favours numeric constants and symbolic terms without variables and the answer constraint (with respect to the variables E_1, E_2, D) resulting from the reflection is

$$C(D, E_1, E_2) : E_1 \leq \epsilon/2 \wedge E_2 \leq \epsilon/2 \wedge D \leq \delta_1 \wedge D \leq \delta_2$$

The planner instantiates the meta-variable $\mathcal{C}$ by C. Later The constraint solver searches for terms to instantiate D, E_1, E_2, and $\mathbf{M}$ respectively and then the planner can instantiate these variables everywhere in the proof plan. For LIM+ the instantiation of the implicitly existentially quantified variables is $[D \ / \ \delta = min(\delta_1, \delta_2)]$, $[E_1 \ / \ \epsilon_1 = \epsilon/2]$, $[E_2 \ / \ \epsilon_2 = \epsilon/2]$.

4.4 Results

We successfully planned all the challenge problems of Woody Bledsoe, i.e., the limit theorems LIM+, LIM-, LIM*, the theorems ContinuousComp, Continuous+, Continuous-, Continuous*, $\lim_{x \to a} x = a$, $\lim_{x \to a} c = c$, and many theorems about limits of polynomial functions lke $\lim_{x \to a} x^2 = a^2$.

A high-level proof plan for LIM+ is shown in Figure 5, where the dashed parts represent those methods/subplans that are planned later at a lower hierarchical level. Some of the subgoals (and corresponding methods) are hidden in the method `UnwrapHyp` that produced subgoals for the next lower level, in particular several inequalities that are satisfied by `Solve-b` or `Solve*`. In the end the expanded plan has 215 nodes.

Among the proven theorems are several that are beyond the capabilities of traditional systems, e.g., LIM* and ContIfDeriv. One reason why LIM* is more complicated to prove than LIM+ becomes clear from its plan in Figure 6. The method $\texttt{ComplexEstimate}_{<}$ is applied once only in the plan for LIM+, whereas for LIM* this trick has to be applied three times because the linear decomposition of the term $f(x) * g(x) - (l_1 * l_2)$ yields more complicated subgoals than the decomposition of $f(x) + g(x) - (l_1 + l_2)$. Figure 6 shows a screen dump of the plan for LIM* that was automatically generated by the ΩMEGA system. The circles indicate nodes of the plan containing methods and subgoals, the squares indicate coreferences, and the triangles indicate assumptions and hypotheses in the proof (they are differentiated by colors, hence not visible in this black and white print); each icon can be clicked on to display the formula and method it stands for.

A comparison of the search spaces of our proof planning system and the automated theorem prover, OTTER, shows some interesting characteristics.

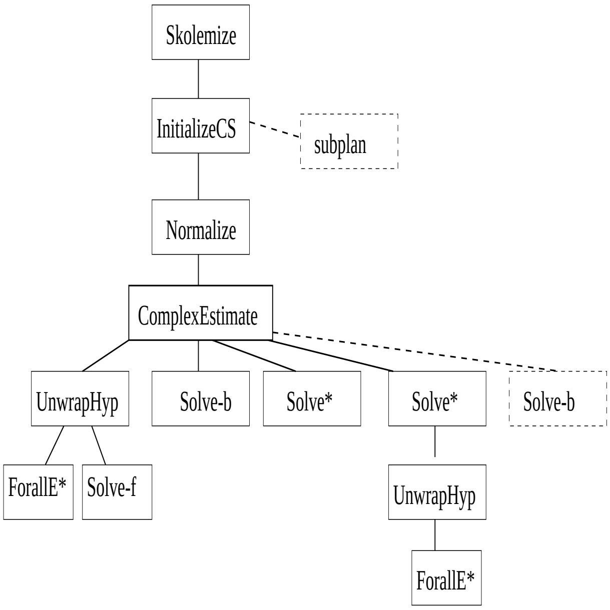

Figure 5: Unexpanded proof plan for LIM+

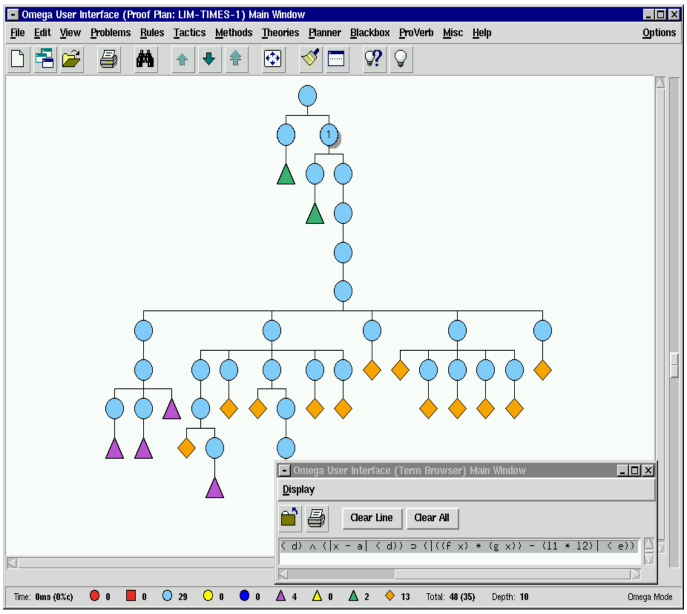

Figure 6: LIM* proof plan as presented by the ΩMEGA interface

theorem	ΩMEGA (matching attempts)	OTTER (generated clauses)/mode
LIM+	346	3650 /spec heuristic for LIM+
LIM+	346	– / auto mode
LIM*	467	– / any
ContIfDeriv	490	– / any

For a simple version of LIM+ and with a particular strategy, OTTER generates a search space of 3650 clauses and keeps 1418 clauses.[13] This strategy is tailored to LIM+ and does not work for LIM* or other limit theorems. In auto-mode OTTER generates up to 437,706,898 clauses and does not succeed in proving LIM+. In comparison, the search space of our planner is about 350 matching attempts for LIM+.[14]

Currently, we are looking at all the theorems, examples, and exercises in two chapters of [9] about limits of sequences and of functions and about continuity. So far we have found that even more theorems can be proved with the methods and control-rules decribed above, e.g., the theorem that says: if $f(x)$ converges to zero and the magnitude of $g(x)$ has an upper bound, then $f(x)*g(x)$ converges to zero ($\lim\limits_{x\to a} f(x) = 0 \wedge \exists y. |g(x)| < y \to \lim\limits_{x\to a} f(x)*g(x) = 0$) or the Squeeze Theorem (For sequences (x_n), (y_n), and (z_n) of real numbers with $x_n \leq y_n \leq z_n$ for all $n \in \mathbb{N}$ holds that if $\lim(x_n) = \lim(z_n)$, then (y_n) is convergent and $\lim(x_n) = \lim(y_n) = \lim(z_n)$ (Theorem 3.2.7 in [9])). For some other examples we need additional facts about the particular functions involved, e.g., for trigonometric functions we need to know the laws of trigonometry. Of course, in order to find proof plans for all the theorems, examples, and exercises in the two chapters, we have to introduce some general methods like `Indirect` and a few more methods for estimation, e.g., `FactorialEstimate`, `EnvironmentEstimate`, and `ComplexEstimate`$_>$. The latter method is needed, for instance, for proving the theorem LIMdiv and the theorem about the uniqueness of a limit.[15] This indicates that a couple of dozen methods and control rules really will capture *all* the mathematical knowledge that appears to be necessary to master this albeit small branch of mathematics. With new methods the control knowledge has to be extended too.

Some of the examples of these two chapter of [9] cannot be proof planned automatically with our current repertoire of techniques, e.g., theorem 4.1.8, exercise 4.1.(3), and exercise 4.1.(12). The reason is that they would need a strategy for eagerly instantiating variables that is flexibly controlled. In addition, with the progress in a textbook, more methods become available that employ the theorems already proved. For instance, the application of the actual limit theorems LIM+, LIM* etc. can greatly facilitate the solution of examples and exercises in the chapters as compared with ϵ-δ-proofs. This means that we can proof plan with different sets of methods. These problems gave rise to further develop our proof planning approach to a multi-strategy planning [89, 90].

Currently, the planner, the domain, and $\mu\mathcal{CAS}$ are implemented in Common LISP (CLOS) while the **l**ovely Ω **u**ser **i**nterface, LΩUI, as well as the constraint solver are implemented in the concurrent logic programming language Mozart-Oz [113].

[13] using Bill McCune's specific input file.

[14] This plan can still be optimized.

[15] This theorem is formalized by: $\lim\limits_{x\to a} f(x) = l_1 \wedge \lim\limits_{x\to a} f(x) = l_2 \to l_1 = l_2$.

5 The Use of Proof Plans for Proof Presentation

Well known from paradigm shifts [69] in physics, for example, is that problems that are puzzling at best or outright unsolvable in the old paradigm, suddenly fall into place like in a beautiful jigsaw puzzle. In the case of proof planning, this seems to apply at least to proof presentation and proof by analogy as well.

The output of traditional automated theorem proving systems lists a sequence of calculus-level steps, such as resolution and paramodulation. These 'proofs' are hardly readable, let alone intuitively understandable as a mathematical proof. A comprehensible explanation is essential, however, at least for interactive and (semi)automated theorem proving systems that could possibly provide a basis for a tutor or assistant system.

This problem was recognized long ago inter alia by Peter Andrews [4], Frank Pfenning [101], Dale Miller [95], Christoph Lingenfelder [75], Alan Robinson [106], Huang and Siekmann [54] and others and several attempts have been made to produce proof presentations based on ND-rules. In fact, proof presentation became a research topic in its own right [2]. More recently, the presentation of proofs (found by traditional automated theorem proving systems) in natural language has been realized in ILF [32], Theorema [20], and PROVERB [53]. ILF provides a schematic verbalization, whereas PROVERB abstracts the calculus level proof first to the so-called assertion-level[16] and then employs linguistic knowledge in order to combine single assertion-level steps into a more coherent natural language presentation.

Nevertheless, there was no satisfying general approach to a *comprehensible* proof presentation so far. We think that this is due to the fact that all automated proof presentation up to now is based on representations of proofs found by traditional theorem proving systems whose level of abstraction is too low and too far removed from the mathematical theory and structures the proof is formulated in. Even the verbalization at the somewhat abstract assertion-level is not necessarily the most natural and best way to communicate a proof to mathematicians or students. Verbalized proof steps can be far *more abstract* than assertion-level steps and may contain explanations too. For example, proofs of limit theorems in R.G. Bartle and D.R. Sherbert's book [9] contain phrases like "We need to estimate the magnitude of..." and there may also be explanations on why a certain step was chosen in the proof.

Now, the abstraction level(s) contained in proof *plans* provide the basis for a truly hierarchical presentation and the design of domain-dependent and -independent methods leads naturally to the design of verbalization schemes that reflect mathematical practice and standard. Furthermore, the subproofs contributed by domain-specific reasoners such as constraint solvers can be easily isolated and then represented separately in the final presentation.

A full, still abstract, verbalization of the LIM+ plan that uses a schematic verbalization of methods, is the following.[17]

[16] An assertion is an axiom or definition the proof refers to.

[17] The occurrence of M is due to the more general presentation needed for other limit theorems. It is, strictly speaking, not necessary for the LIM+ verbalization. The itemization would not occur in a textbook but is used here to show the correspondence between verbalization and proof plan.

To show that $f(x) + g(x)$ converges to $L1 + L2$

(*) Let δ be smaller than δ_1, δ be smaller than δ_2, and ϵ_1, ϵ_2 be smaller than $\epsilon/2$.

1. We need to estimate the magnitude of $|f(x) + g(x) - (L1 + L2)| = |(f(x) - L1) + (g(x) - L2)|$.

2. To do this, we use the Triangle Inequality and obtain $|f(x) + g(x) - (L1 + L2)| \leq |f(x) - L1| + |g(x) - L2|$.
   This goal can be shown in three steps:

   - There exists an M such that $|1| \leq M$ and
   - $|f(x) - L1| < \epsilon/(2 * M)$, and
   - $|g(x) - L2| < \epsilon/2$.

3. For a real number $1 \leq M$

4. by hypothesis, if $\epsilon/(2 * M) > 0$, there exists δ_1 such that for all x, if $|x - a| < \delta_1$ then $|f(x) - L1| < \epsilon/(2 * M)$

5. by hypothesis if $\epsilon/2 > 0$, there exists δ_2 such that forall x, if $|x - a| < \delta_2$, then $|g(x) - L2| < \epsilon/2$.

6. From (*) follows that

7. if $|x - a| < D$, then $$\begin{aligned} |f(x) + g(x) - (L1 + L2)| &\leq M * |f(x) - L1| + |g(x) - L2| \\ &< M * \epsilon/(2 * M) + \epsilon/2 = \epsilon \end{aligned}$$
   and therefore $|f(x) + g(x) - (L1 + L2)| < \epsilon$.

8. Since $\epsilon > 0$ is arbitrary,

9. the theorem is proven. qed

Every item in the above verbalization corresponds directly to (parts of) one of the methods in Figure 5. Not every method is verbalized and `ComplexEstimate` has several verbalization parts. In more detail, the steps 1, 2, and 7 above are generated as a verbalization of `ComplexEstimate`; steps 3, 4, and 5 verbalize subproofs of `ComplexEstimate`'s subgoals, and step (*) verbalizes `InitializeCS` by the answer constraint formula.

Currently, the verbalization of single methods and of whole proof plans is automatically generated and presented in a hypertext window of ΩMEGA 's interface $\mathcal{L}\Omega\mathcal{UI}$ [88, 92] as shown in Figure 7 in the bottom right LMLB window. The local hypertext verbalization, i.e., the verbalization of a method, corresponds to the nodes (methods) in the plan that are clicked on. Proofs of subgoals produced by the method are linked by hyperlinks. The global verbalization of a whole plan can still be improved considerably. Linguistic knowledge has to be employed and for combining the verbalization of several methods into a nice linear presentation of the whole proof plan as in the automatic verbalization of the assertion-level proofs [53] discussed above.

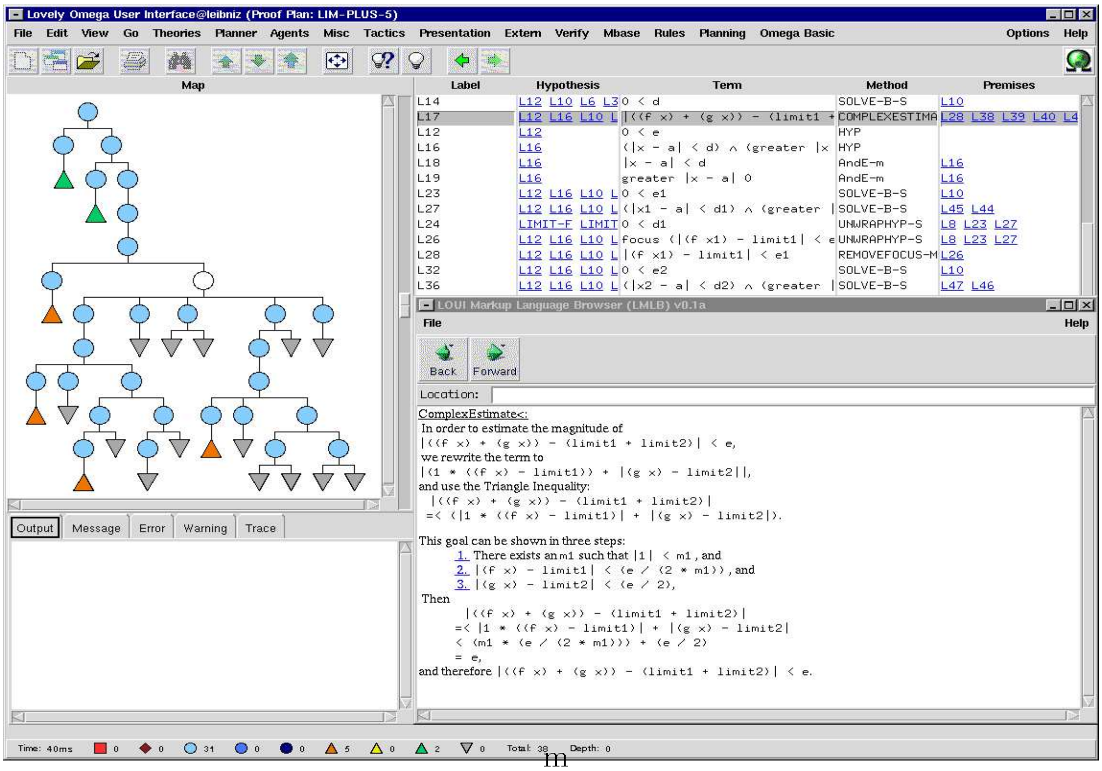

Figure 7: Screen shot of ΩMEGA showing a proof plan of LIM+ and a local verbalization

6 Conclusion

Automated theorem proving based on proof planning was motivated by the growing disillusionment with traditional ATP.

This article introduces *knowledge-based* proof planning that borrows techniques and formalisms from many branches of AI, such as hierarchical planning, knowledge representation in frames and explicitly represented control-rules, constraint solving, search-based deduction systems as well as tactical and meta-level reasoning. We use a general-purpose planner and we encode the mathematical domain knowledge, as we may find it in a graduate textbook on mathematics, into methods, theory-specific reasoners, and control-rules.

Control-rules make the control far more flexible and considerably extend the *meta-level reasoning* facilities of proof planning that are used for global guidance. This declaratively represented control knowledge can express conditions for a decision that depends on the current planning state, the planning history, failed proof attempts, the current partial proof plan, the constraint state, the available resources, the user model, the theory in which to plan, typical models of the theory, etc. and the solution of the more difficult problems would have been impossible without it.

In contrast to conventional planning domains, a proof planning domain is represented by mathematical theories, such as *group theory* or *calculus*, that contain as usual axioms, definitions, theorems, and lemmata but also operators, control-rules, and domain-specific external reasoners. The operators, called methods, represent natural mathematical proof steps and the control-rules encode mathematical knowledge on how to proceed when searching for a proof.

For a well known mathematical domain, the limit theorems which we used as an example to serve the need of demonstration, we have collected the relevant mathematical knowledge and represented it in methods and control rules, and also used existing representations of a constraint solver and a computer algebra system. Based on this knowledge, ΩMEGA 's general-purpose proof planner was able to prove more difficult than other current proof planners and than traditional theorem proving systems.

We can potentially find proofs of a length far beyond those a traditional search-based system can generate. To see this, let us compare some statistics by their order of magnitude. Todays traditional theorem proving systems, such as OTTER or SPASS can search spaces of several billion clauses. The maximal proof length that can be found that way is around several hundred resolution or paramodulation steps. With proof planning we could potentially find a *plan* of several hundred steps that could then be expanded into a calculus-level proof of several thousand steps. Proofs of that length are not uncommon for difficult mathematical theorems but also arise in industrial applications, e.g., in program verification tasks. For instance, the VSE verification system [58], which is now routinely used for industrial applications in the German Centre for Artificial Intelligence Research (DFKI) in Saarbrücken, interactively synthesized proofs of up to 8,000 and 10,000 steps for some difficult assertions in the verification of a television and radio switching program. These proofs which often represent several weeks of labor with the system, are, by their very length, one or two orders of magnitude beyond fully automated methods but could come into the range of possibilities, if the proof planning paradigm turns out to be successful in these settings as well.

However, our main interest right now is more in everyday mathematics. Sacrificing the hope that a traditional theorem proving engine based on search at the calculus-level can ever evolve into a mathematical assistant system, gives way to an alternative within which traditional automated theorem proving is a rather small but still useful subtask.

Using the well known albeit exiguous mathematical field of limit theorems, we have been able to show that realistic mathematics can indeed be carried out on a machine. Metaphorically speaking, we have shown the atom can be split and indeed it gives off energy – but as a show case for a generally useful device and its everyday application the test case is still little representative.

Therefore, several PhD students are currently extending our knowledge needed in proof planning to fields such as linear algebra, analysis, and finite group theory. In particular, we are planning to set up a distinguished international consortium of mathematicians, computer scientists, and some interested companies to encode significantly broad areas of mathematics into MBase and to use this knowledge for proof planning with ΩMEGA. We believe that the extraction and explicit representation of the knowledge of wide mathematical fields will – just like the motivation for CYC – ultimately be useful not only for computer-supported mathematics but also for mathematical education systems.

6.1 Related Work

This work has been deeply influenced by the work of Woody Bledsoe. The knowledge acquisition for the design of methods for limit proof plans is similar to the ideas in the special-purpose theorem prover IMPLY [18] and to Beeson's work [11] whose bias is, however, more towards special-purpose provers. Beeson's ϵ-δ-proofs with the *Mathpert* and *Weierstrass* systems were developed in parallel to our's [84].

More generally, our approach has the use of theory-specific knowledge in common with other special-purpose theorem provers, such as a system for monoids [40], for geometry, or for set theory [50].

Closest to our work, is the proof planning approach developed by Alan Bundy for the *CLAM* system. As opposed to our (domain-dependent) control knowledge represented in control-rules, *CLAM* uses rippling, a domain-independent difference reduction heuristic which is encoded in the preconditions of methods. LIM+ was proved in *CLAM* with colored rippling [118], an extension of rippling but LIM* and other theorems turned out to be too difficult for *CLAM*.

As for the integration of external reasoners into automated theorem proving systems, several attempts have recently been made for integrating CAS's (cf. [8, 46, 30, 64]) and few attempts of integrating constraint solving [116] into ATP.

Our work relates to research in planning, e.g. in Prodigy [97], that uses control-rules. Weld [122] and Veloso et al. [120] discuss the superiority of knowledge-based search with a production system of forward-chaining rules over implicitly encoded control, e.g., by functional rating. Such control-rules were first explored in SOAR [70] and then in Prodigy [97]. Gerberding and Noltemeier [42] formulate program-like rules for strategies in theorem proving by mathematical induction.

Very recently, there has been a related approach for presenting proofs that verbalizes proofs *manually* found with the tactical prover Nuprl [51]. It also verbalizes proofs at a more abstract level.

Last but not least, the presented work owes to other developments in the ΩMEGA group, in particular, the design of methods [56], the integration of computer algebra systems [64], and analogy-driven proof plan construction [83, 94].

7 Acknowledgement

This work builds upon the paradigm change in the work of the ΩMEGA group in Saarbrücken and the discussion over many years that led to it are gratefully acknowledged. Last but not least, the implementation builds upon the ΩMEGA system whose development was supported by many people [12].

Furthermore we would like to thank Bob Boyer for his help with the IMPLY prover when our project started, and in particular Alan Bundy and the DREAM group in Edinburgh for many discussions and their influence. We thank Jaime Carbonell and Manuela Veloso for discussions about control-rules and Martin Müller for his help to getting started with constraint solving. Last but not least we thank my students Carsten Ullrich and Jürgen Zimmer who implemented much of the knowledge-based proof planning and kept abreast of the many changes of the ΩMEGA system while this work lasted.

References

[1] *Human Oriented Proof Presentation: A Reconstructive Approach.* INFIX Publ.Comp, 1996.

[2] *First International Workshop on Proof Transformation and Presentation*, Dagstuhl, 1997.

[3] R. Anderson and W.W. Bledsoe. A linear proof format for resolution with merging and a new technique for establishing completeness. *Journal of the ACM*, 17(3):525–534, July 1970.

[4] P.B. Andrews. Transforming matings into natural deduction proofs. In *Proceedings of the 5th International Conference on Automated Deduction*, pages 281–292. Springer, 1980.

[5] P.B. Andrews. Theorem proving via general matings. *Journal of ACM*, 28:193–214, 1981.

[6] F. Baader and J. Siekmann. *Handbook of Logic in Artificial Intelligence*, chapter Unification Theory. Oxford University Press, 1997.

[7] A.M. Ballantyne and W.W. Bledsoe. Automatic proofs of theorems in analysis using non-standard techniques. *JACM*, 24:353–374, 1977.

[8] C. Ballarin, K. Homann, and J. Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In A. H. M. Levelt, editor, *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'95)*, pages 150–157, Berkeley, CA, USA, 1995. ACM Press.

[9] R.G. Bartle and D.R. Sherbert. *Introduction to Real Analysis.* John Wiley& Sons, New York, 1982.

[10] P. Baumgartner and U. Furbach. PROTEIN: A prover with a theory extension interface. In A. Bundy, editor, *Proceedings of the 12th Conference on Automated Deduction*, volume 814 of *Lecture Notes on Artificial Intelligence*, pages 769–773, Nancy, 1994. Springer.

[11] M. Beeson. Automatic generation of epsilon-delta proofs of continuity. In J. Calment and J. Plaza, editors, *Artificial Intelligence and Symbolic Computation*, volume 1476 of *Lecture Notes on Artificial Intelligence*, pages 67–83, 1998. Springer.

[12] C. Benzmueller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. OMEGA: Towards a mathematical assistant. In W. McCune, editor, *Proceedings 14th International Conference on Automated Deduction (CADE-14)*, pages 252–255, Townsville, 1997. Springer.

[13] W. Bibel. On matrices with connections. *Journal of the ACM*, 28:633–645, 1981.

[14] W.W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977.

[15] W.W. Bledsoe. Some thoughts on proof discovery. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 2–10, Salt Lake City, UT, 1986.

[16] W.W. Bledsoe. The use of analogy in automatic proof discovery. Tech.Rep. AI-158-86, Microelectronics and Computer Technology Corporation, Austin, TX, 1986.

[17] W.W. Bledsoe. Challenge problems in elementary analysis. *Journal of Automated Reasoning*, 6:341–359, 1990.

[18] W.W. Bledsoe, R.S. Boyer, and W.H. Henneman. Computer proofs of limit theorems. *Artificial Intelligence*, 3(1):27–60, 1972.

[19] D. Borrajo and M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 10:1–34, 1996.

[20] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. An overview of the theorema project. In *Proceedings of ISSAC'97*, 1997.

[21] A. Bundy. Doing arithmetic with diagrams. In *Adv. Papers Thrid International Joint Conference on Artificial Intelligence*, pages 130–138, 1973.

[22] A. Bundy. The use of explicit plans to guide inductive proofs. In E. Lusk and R. Overbeek, editors, *Proc. 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120, Argonne, 1988. Springer.

[23] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[24] A. Bundy, F. van Harmelen, A. Ireland, and A. Smaill. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction CADE'90*, volume 449 of *LNCS*. Springer, 1990.

[25] H.-J. Bürckert. A resolution principle for constrained logics. *Artificial Intelligence*, 66(2):235–271, 1994.

[26] J.G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371–392. Morgan Kaufmann Publ., Los Altos, 1986.

[27] Weidenbach Ch. Spass: Version 0.49. *Journal of Automated Reasoning, Special Issue on the CADE-13 Automated Theorem Proving System Competition*, 18(2):247–252, 1997.

[28] L. Cheikhrouhou and J. Siekmann. Planning diagonalization proofs. In *Proceedings of 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA'98)*, volume 1480 of *Lecture Notes on Artificial Intelligence*, Sozopol, Bulgaria, 1998.

[29] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[30] E. Clarke and X. Zhao. Analytica-A Theorem Prover in Mathematica. In *11th Conference on Automated Deduction*, pages 761–763. Springer, 1992.

[31] 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. Panagaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[32] B.I. Dahn and A. Wolf. Natural language presentation and combination of automatically generated proofs. In *Proceedings of FroCos'96*, pages 175–192. Kluwer, 1996.

[33] M. Davis. A computer program for Presburger's algorithm. In A. Robinson, editor, *Proving Theorems (as done by Man, Logician, or Machine)*, pages 215–233. Cornell University, 1957. Summary of talks presented at the 1957 summer institute for symbolic logic.

[34] M.D. Davis, R. Sigal, and E.J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, second edition, 1994.

[35] W. Davis, H. Porta, and J. Uhl. *Calculus & Mathematica*. Addison-Wesley, 1994.

[36] N. Eisinger and H.J. Ohlbach. The Markgraf Karl Refutation Procedure (MKRP). In J.Siekmann, editor, *Proc. 8th International Conference on Automated Deduction (CADE)*, volume 230 of *LNCS*, pages 681–682, Berlin, 1986. Springer.

[37] G. Faltings and U. Deker. Interview: Die Neugier, etwas ganz genau wissen zu wollen. *bild der wissenschaft*, (10):169–182, 1983.

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

[39] A. Franke and M. Kohlhase. MBase: representing mathematical knowledge in a relational database. In A. Armando and T. Jebelean, editors, *Calculemus'99*, pages 135–152, 1999.

[40] H. Ganzinger and U. Waldmann. Theorem proving in cancellative abelian monoids. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes on Artificial Intelligence*, pages 388–402. Springer, 1996.

[41] Gerhard Gentzen. Untersuchungen über das logische Schließen I& II. *Math. Zeitschrift*, 39:176–210,572–595, 1935.

[42] S. Gerberding and A. Noltemeier. Incremental proof-planning by meta-rules. In *Proceedings of the 10th Florida International AI Conference (FLAIRS-97)*, Dayton Beach, FL, 1997.

[43] M. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. volume 78 of *Lecture Notes in Computer Science*. Springer, Berlin, 1979.

[44] P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes on Artificial Intelligence*. Springer, 1996.

[45] J. Hadamard. *The Psychology of Invention in the Mathematical Field*. Princeton Univ. Press, Princeton, 1945.

[46] J. Harrison and L. Théry. Reasoning About the Reals: The Marriage of HOL and Maple. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR'93)*, volume 698 of *Lecture Notes on Artificial Intelligence*, pages 351–353, St. Petersburg, 1993. Springer.

[47] P. Hayes and D.B. Anderson. An Arraignment of Theorem-Proving; or, The Logician's Folly. Memo 54, Dept. Computational Logic, Edinburgh University, 1972.

[48] C. Hewitt. Description and theoretical analysis of PLANNER. Technical report, 1972.

[49] T. Hillenbrand and A. Buch. Waldmeister: Development of a high performance completion-based theorem prover. Seki Report SR-96-01, Universität Kaiserslautern, 1996.

[50] L.M. Hines. Str+ve$\subset$: The stri+ve-based subset prover. In *Proceedings of the Tenth Conference on Automated Deduction (CADE)*, 1990.

[51] A.M. Holland-Minkley, R. Barzilay, and R.L. Constable. Verbalization of high-level formal proofs. *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, 1999.

[52] Ch. Holzbaur. OFAIclp(q,r) manual. Technical Report TR-95-09, Austrian Institute for Artificial Intelligence, Wien, 1995.

[53] X. Huang and A. Fiedler. Proof verbalization as an application of nlg. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence*, pages 965–970. Morgan Kaufmann, 1997.

[54] X. Huang and J. Siekmann. *Proof presentation* Springer, to appear.

[55] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. Omega-MKRP: A proof development environment. In *Proceedings of the 12th International Conference on Automated Deduction (CADE)*, Nancy, 1994.

[56] X. Huang, M. Kerber, M. Kohlhase, and J. Richts. Methods - the basic units for planning and verifying proofs. In *Proceedings of Jahrestagung für Künstliche Intelligenz KI-94*, Saarbrücken, 1994. Springer.

[57] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *Proceedings of 10th International Conference on Automated Deduction (CADE-10)*, volume 449 of *Lecture Notes in Artificial Intelligence*. Springer, 1990.

[58] D. Hutter, B. Langenstein, C. Sengler, J.H. Siekmann, W. Stephan, and A. Wolpers. Deduction in the verification support environment (VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings of the Third International Symposium of Formal Methods Europe*, pages 268–286, Oxford, England, 1996.

[59] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[60] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[61] J. Jaffar, S. Michaylow, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages*, 14(3):339–395, 1992.

[62] S. Kambhampati. Refinement planning: Status and prospectus. In *Proceedings of the AAAI-96*, pages 1331–1336. Morgan Kaufmann, 1996.

[63] S. Kambhampati and B. Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 61–78. IOS Press, Amsterdam, Oxford, 1996.

[64] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra into Proof Planning. *Journal of Automated Reasoning*, 1998. forthcoming.

[65] M. Kerber and E. Melis. Using exemplary knowledge for justified analogical reasoning. In Michel De Glas and Zdzislaw Pawlak, editors, *WOCFAI '95 – Proceedings of the Second World Conference on the Fundamentals of Artificial Intelligence*, pages 157–168, Paris, France, 1995. Angkor.

[66] M. Kerber, E. Melis, and J. Siekmann. Reasoning with assertions and examples. In *AAAI Spring Symposium on Artificial Intelligence and Creativity*, pages 61–66, 1993.

[67] K.R. Koedinger and J.R. Anderson. Abstract planning and perceptual chunks: Elements of expertise in geometry. *Cognitive Science*, 14:511–550, 1990.

[68] G. Kreisel. Mathematical logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons, 1965.

[69] T.S. Kuhn. *Die Struktur wissenschaftlicher Revolutionen.* Suhrkamp in German, Frankfurt/M., 1976.

[70] J Laird, A. Newell, and P. Rosenbloom. SOAR:an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[71] C. Leckie and I. Zukerman. Inductive learning of search control rules for planning. *Artificial Intelligence*, 101(1-2):63–98, 1998.

[72] U. Leron. Heuristic presentations: the role of structuring. *For the Learning of Mathematics*, 5(3):7–13, 1985.

[73] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[74] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[75] Ch. Lingenfelder. *Transformation and Structuring of Computer Generated Proofs.* PhD thesis, University Kaiserslautern, 1990.

[76] D. Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, New York, 1978.

[77] H. Lüneburg. *Vorlesungen über Analysis.* BI Wissenschaftsverlag, 1981.

[78] S. MacLane. *Mathematics: Form and Function.* Springer, 1986.

[79] W. McCune. 33 basic test problems: A practical evaluation of some paramodulation strategies. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 5, pages 71–114. MIT Press, 1997.

[80] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997. to appear.

[81] W.W. McCune. Otter 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, Maths and CS Division, Argonne, Illinois, 1990.

[82] E. Melis. Analogies between proofs – a case study. SEKI-Report SR-93-12, University of Saarbrücken, Saarbrücken, 1993.

[83] E. Melis. A model of analogy-driven proof-plan construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 182–189, Montreal, 1995.

[84] E. Melis. Progress in proof planning: Planning limit theorems automatically. Technical Report SR-97-08, Fachbereich Informatik, Universität des Saarlandes, October 1997.

[85] E. Melis. The Heine-Borel challenge problem: In honor of Woody Bledsoe. *Journal of Automated Reasoning*, 20(3):255–282, 1998.

[86] E. Melis. The "limit" domain. In R. Simmons, M. Veloso, and S. Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence in Planning Systems (AIPS-98)*, pages 199–206, 1998.

[87] E. Melis. AI-techniques in proof planning. editors, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, Kluwer, pages 494–498, 1998.

[88] E. Melis. Proof presentation based on proof plans. Seki report SR-98-08, Universität des Saarlandes, FB Informatik, November 1998.

[89] E. Melis. Proof planning with multiple strategies. In *CADE-15 workshop: Strategies in Automated Deduction*, 1998.

[90] E. Melis and A. Meier. Proof Planning with Multiple Strategies II. In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *FLoC'99 workshop on Strategies in Automated Deduction*, 1999.

[91] E. Melis and V. Sorge. Employing External Reasoners in Proof Planning. In A. Armando and T. Jebelean, editors, *Calculemus'99*, pages 123–134, 1999.

[92] E. Melis and U. Leron. A proof presentation suitable for teaching proofs. In S.P. Lajoie and M. Vivet (eds.), *9th International Conference on Artificial Intelligence in Education*, IOS Press, pages 483-490, 1999.

[93] E. Melis and J.H. Siekmann. Concepts in proof planning. In *Intellectics and Computational Logic. Papers in Honor of Wolfgang Bibel*, pages 249–264. Kluwer, 1999.

[94] E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2):117–147, 1999.

[95] D. Miller. *Proofs in Higher Order Logic.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1983.

[96] S. Minton. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[97] S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. *PRODIGY 2.0: The Manual and Tutorial.* School of Computer Science, Carnegie Mellon University, Pittsburgh, 1989. CMU-CS-89-146.

[98] 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.

[99] H. De Nivelle. *Bliksem User Manual.* Delft University of Technology, 1998.

[100] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–368. Academic Press, 1990.

[101] F. Pfenning. *Proof Transformation in Higher Order Logic.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1987.

[102] G. Polya. *How to Solve it.* Princeton University Press, Princeton, 1945.

[103] D. Prawitz. *Natural Deduction - A proof theoretical study.* Almquist and Wiksell, Stockholm, 1965.

[104] R. Reiter. A semantically guided deductive system for automatic theorem proving. In *Third International Joint Conference on Artificial Intelligence*, pages 41–46, 1976.

[105] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12, 1965.

[106] J.A. Robinson. Proof = guarantee + explanation. In *Intellectics and Computational Logic. Papers in Honor of Wolfgang Bibel.* Kluwer, 1998.

[107] Russell and Whitehead. *Principia Mathematica.* -, -.

[108] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[109] A.H. Schoenfeld. *Mathematical Problem Solving.* Academic Press, New York, 1985.

[110] J. Siekmann. Unification theory: A survey. *Symbolic Computation*, 3,4(7), 1989.

[111] J.H. Siekmann, M. Kohlhase, and E. Melis. Omega: Ein mathematisches Assistenzsystem. *Kognitionswissenschaft*, 1998.

[112] J. Siekmann, S. Hess, Ch. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, and V. Sorge. $\mathcal{L}\Omega\mathcal{UI}$: Lovely ΩMEGA User Interface. *Formal Aspects of Computing*, 3:1–18, 1999.

[113] G. Smolka. *An Oz Primer.* Programming System Lab, DFKI Saarbrücken, 1995.

[114] G. Smolka. The Oz programming language. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, Berlin, 1995.

[115] R.M. Smullyan. *First-Order Logic.* Springer, 1968.

[116] F. Stolzenburg. Membership constraints and complexity in logic programming with sets. In F. Baader and U. Schulz, editors, *Frontiers in Combining Systems*, pages 285–302. Kluwer Academic, Dordrecht, The Netherlands, 1996.

[117] A. Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893. Morgan Kaufmann, 1977.

[118] Y. Tetsuya, A. Bundy, I. Green, T. Walsh, and D. Basin. Coloured rippling: An extension of a theorem proving heuristic. In A.G. Cohn, editor, *Proceedings of 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 85–89. John Wiley, 1994.

[119] B.L. van der Waerden. Wie der Beweis der Vermutung von Baudet gefunden wurde. *Abh.Math.Sem.Univ.Hamburg*, 28, 1964.

[120] M. Veloso, J. Carbonell, M. A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 81–120, 1995.

[121] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4:2–22, 1960.

[122] D.S. Weld. An introduction to least committment planning. *AI magazine*, 15(4):27–61, 1994.