Progress in Proof Planning: Planning Limit Theorems Automatically

Erica Melis
Universität des Saarlandes, Fachbereich Informatik,
D-66041 Saarbrücken, Germany,
melis@cs.uni-sb.de

Contents

1 Introduction **2**
1.1 The Class of Limit Theorems . . . 3
1.2 Objectives of the Report . . . 4

2 Brief Review of Proof Planning **5**
2.1 Proof Planning in CL^AM . . . 6
2.2 Original Proof Planning in OMEGA . . . 8

3 Our Extensions of OMEGA: Domain Knowledge and Architecture **11**
3.1 Methods for Planning Limit Theorems . . . 13
3.1.1 Basic Methods . . . 13
3.1.2 Supermethods . . . 20
3.2 Combination of Proof Planning with Constraint Solving . . . 23
3.2.1 General Framework for Constraint Solving . . . 24
3.2.2 Combining Proof Planning with Constraint Solving . . . 25
3.2.3 The Constraint Solver LINEQ . . . 25
3.3 Mathematical Control Knowledge . . . 31
3.3.1 Syntax of Control-Rules . . . 31
3.3.2 Control-Rules for Planning Limit Theorems . . . 32

4 Exemplary Proof Planning for LIM+ **35**

5 Extracting a Proof from the Plan **38**

6 Experiments and Results **38**

7 Conclusion and Related Work **40**

8 Acknowledgement **42**

Abstract

Proof planning is an alternative methodology to classical automated theorem proving based on exhausitve search that was first introduced by Bundy [8]. The goal of this paper is to extend the current realm of proof planning to cope with genuinely mathematical problems such as the well-known limit theorems first investigated for automated theorem proving by Bledsoe. The report presents a general methodology and contains ideas that are new for proof planning and theorem proving, most importantly ideas for search control and for the integration of domain knowledge into a general proof planning framework. We extend proof planning by employing explicit control-rules and supermethods. We combine proof planning with constraint solving. Experiments show the influence of these mechanisms on the performance of a proof planner. For instance, the proofs of LIM+ and LIM* have been automatically proof planned in the extended proof planner OMEGA.

In a general proof planning framework we rationally reconstruct the proofs of limit theorems for real numbers ($\mathbf{R}$) that were first computed by the special-purpose program reported in [6]. Compared with this program, the rational reconstruction has several advantages: It relies on a general-purpose problem solver; it provides high-level, hierarchical representations of proofs that can be expanded to checkable ND-proofs; it employs declarative contol knowledge that is modularly organized.

1 Introduction

While humans can cope with long and complex proofs and have strategies to avoid less promising proof paths, traditional automated theorem proving suffers from exhaustive search in super-exponential search spaces. As a potential solution of this problem, proof planning has been introduced by Bundy in [8] as an alternative to the methodology of classical automated theorem proving. It employs high-level planning operators rather than calculus-level rules and global control as opposed to the more local search heuristics which are used for search control in automated theorem proving (see, e.g. [29]).

However, there are classes of theorems that are difficult or impossible to prove by state-of-the-art proof planners such as CL^AM [10] or OMEGA [3]. For instance, the limit theorem LIM* is beyond the capabilities of current theorem provers and proof planners. For many problems, the available control means in these systems do not sufficiently reduce the search or the difficulties are caused by the need to construct mathematical objects with certain properties rather than searching for them. Such constructions occur, for example, in the Gram–Schmidt orthogonalization process [26] and in proving theorems about limits. Therefore, we investigate proof planning for the well-known class of limit theorems, using them as a "Drosophila"[1] of proof planning. There are more reasons why we have chosen the class of limit theorems:

- The limit theorems are well-known.
- Some of the knowledge engineering work was done already in [6]. We encode some of the heuristics employed there in our methods.

[1] Drosophila is a fly that has intensively been investigated in molecular biology in order to find scientific explanations/results for more complex systems too.

- The limit theorems are genuinely mathematical problems that are at the edge or beyond the capabilities of the current fully automated theorem provers of the classical type.

We investigate a **general methodology** with new features of proof planning mathematical theorems that manifest in planning limit theorems. Consequently, we suggest an extended architecture for proof planners. The resulting extensions of OMEGA go beyond the needs for a particular class of theorems. They address the following general tasks.

- Representing domain knowledge explicitly which includes methods, control-rules, and domain-dependent constraint solvers. Here, domain means "mathematical theory", such as set theory, group theory, integers, etc.
- Combining proof planning with constraint solving mechanisms.
- Integrating modularly represented, declarative, high-level control into proof planning.
- Designing planning strategies and complex methods to reduce the search.

Our goal is to rise to Bledsoe's challenge to "pay attention to mathematician's great deal of direction' rather than 'cover the eyes with blinders and hunt through a cornfield for a diamond-shaped grain of corn" issued in 1986 [4].

1.1 The Class of Limit Theorems

The class of limit theorems includes the well-known theorem LIM+ from calculus that states that the limit of the sum of two functions in $\mathbb{R}$ is the sum of their limits. Other theorems in this class are, e.g., LIM- and LIM*, similar theorems about differences and products, COMPOSITE that states that the composition of two continous functions is continous, CONTINOUS that states that a function having a derivative at a point is continuous there, and theorems like $\lim_{x \to a} x^2 = a^2$. These theorems are formulated in the theory of the real numbers $\mathbb{R}$. One of the simplest of these theorems, LIM+, is proposed as a challenge problem in [5] and given next.

$$\lim_{x \to a} f(x) = L_1 \wedge \lim_{x \to a} g(x) = L_2 \rightarrow \lim_{x \to a} f(x) + g(x) = L_1 + L_2, \tag{1}$$

which, after expansion of the definition of lim, is
$\forall \epsilon_1 \exists \delta_1 \forall x_1 (0 < \epsilon_1 \rightarrow 0 < \delta_1 \wedge |x_1 - a| < \delta_1 \rightarrow |f(x_1) - L_1| < \epsilon_1) \wedge$
$\forall \epsilon_2 \exists \delta_2 \forall x_2 (0 < \epsilon_2 \rightarrow 0 < \delta_2 \wedge |x_2 - a| < \delta_2 \rightarrow |f(x_2) - L_1| < \epsilon_2)$
$\rightarrow \forall \epsilon \exists \delta \forall x (0 < \epsilon \rightarrow 0 < \delta \wedge |x - a| < \delta \rightarrow |(f(x) + g(x)) - (L_1 + L_2)| < \epsilon)$.

The typical way a mathematician goes about to prove such a theorem is to (incrementally) invent an instantiation of δ that depends on ϵ, δ_1, and δ_2. This is a non-trivial thing to do and difficult for students as reported in [26]. One way to construct δ is described by Bartle and Sherbert in their textbook [2]. They constrain the number n (that correseponds to δ) in the proof for products of limits of the sequences $X = (x_n)$ and $Y = (y_n)$ by using intermediately introduced variables M_1, M, K_1, and K_2[2] upon which n depends. "According

[2] K_1, K_2 correspond to δ_1, δ_2 in LIM+

to Theorem 3.2.2. 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

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

From the convergence of X and Y 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

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

Since ϵ is arbitrary, this proves that the sequence $X * Y$ converges to $x * y$" [2].

Inspired by a similar mathematical idea, Bledsoe, Boyer, and Henneman presented a *special-purpose* theorem prover for limit theorems, IMPLY [6]. It attempts to prove formulae of the form $|A| < \epsilon_1 \vdash |B| < \epsilon$ by representing B as a linear combination of A, $B = k * A + l$, and by proving the simpler formulae $|k| < M$, $|A| < \frac{\epsilon}{2M}$, and $|l| < \frac{\epsilon}{2}$, containing a new variable M. The IMPLY prover employed this heuristic and a number of other rules to restrict the value of the variable δ. It seems that this work is not as well recognized nowadays in the field – maybe because of its special-purpose character.

1.2 Objectives of the Report

Planning inductive proofs with the proof planner CL^AM [10] and the first planning attempts in OMEGA [3], have been steps towards proof planning. Our goal is to extend the general methodology and the realm of proof planning and to show a more general direction.

In order to demonstrate the potential of extended proof planning, the report shows how to automatically plan limit theorems, i.e., theorems that are at the edge and beyond the capabilities of other current automated systems. For this purpose, we rationally reconstruct and extend IMPLY heuristics in the proof planning framework by designing frequently used proof planning operators. In addition, we enlarge the domain knowledge and extend the architecture of proof planners in order to make use of declarative control-rules that considerably restrict the search. The domain knowledge is also employed to solve constraints in proof planning.

Compared to Bledsoe's original special-purpose prover, the rational reconstruction exhibits the following advantages

- We use a general methodology, namely proof planning.
- We use a general-purpose proof planner.
- Most of the methods employed belong to the theory *base* or *ordered-field* rather than being specific for the limit-class of theorems. The control makes feasible to pick the right methods from a large collection of methods.
- Proof plans are expandable to Natural Deduction (ND) proofs that can be proof checked.
- As exercised with the reconstruction of Nqthm in proof planning [8], proof planning allows for a more flexible use of methods than special-purpose theorem provers with a fixed sequence of routines, e.g., Nqthm's waterfall.

- Declarative global control knowledge restricts the search.
- Proof planning results in a high-level, hierarchical representation of proofs. The rational reconstruction makes *explicit* the methods and the control knowledge and explicates the structure of limit proofs. This is important for an interactive system (for the user's proof understanding), for learning methods and control knowledge, and for reusing proofs.

This paper is organized as follows. After a brief introduction to the state-of-the-art proof planning, we characterize the extensions of proof planning that give rise to the new architecture of our proof planner. Then we present the knowledge that belongs to the *limit* domain theory – being part of OMEGA's hierarchically structured theory knowledge base. This knowledge comprises methods, a constraint solver, and control knowledge available for proof planning the limit theorems. We then provide details of the proof planning for LIM+. Results of experiments show how the new mechanisms influence the feasibility and performance of proof planning limit theorems. A summary, related, and future work conclude the paper.

In the remainder, we use the following naming **conventions**: `METHODS` names in capital letters denote supermethods, names with the capital initial letter denote `Methods`, and names written in small letters denote `procedures`. As for symbol's meaning, $div, *, +, -, val$ denote the division, multiplication, addition, subtraction, and absolute value function in $\mathbb{R}$, respectively. We use σ for substitutions and abbreviate the result of applying σ to F by F_σ.

2 Brief Review of Proof Planning

Automated theorem proving currently witnesses a change: classical techniques based on search *at the calculus level* are augmented by a knowledge-intensive, more abstract and *high-level planning* of a proof.

Proof planning employs intelligent guidance of proofs and high-level planning operators rather than calculus-level rules. The Edinburgh group pioneered proof planning as a technique by building the proof planner *CLAM* [10]. Yet, *CLAM* has not been sufficiently acknowledged by the theorem proving community, maybe because it is specialized to inductive proofs and by and large did not perform much better than other state-of-the-art theorem provers, e.g., than Nqthm [7]. Many people from the theorem proving community mistakenly perceived proof planning as restricted to the difference reduction technique rippling, but there is more to proof planning indeed, see, e.g. [9].

The basic idea of proof planning is that of classical planning in Artificial Intelligence (AI): Operators represent actions and specify preconditions and effects of the action's application. In STRIPS notation, the effects are captured in add and delete lists [15]. The add-list contains the literals (or more general the elements) that are introduced into a state by the operator's application, and the delete list contains those elements that are deleted from the state by the operator's application. A planning *problem* consists of an initial state and goals. A *solution* is a sequence of actions, i.e., of instantiated operators, that transforms the initial state into a state in which the goals hold. Backward planning starts with the conjecture as an open goal g and with assumptions. The planner searches for an operator[3] Op that

[3] In the following we say operator instead of instantiated operator.

proves g and introduces a node annotated with Op into the plan. The subgoals g_i produced by the application of Op become the new *open* subgoals and g now has the status *closed*. The planner continues to search for operators applicable to one of the open subgoals and terminates if there are no more open goals.

A *cognitive* motivation for proof planning is the fact that mathematicians tend to plan proofs. Several empirical sources [32, 1, 24] provide evidence that mathematicians use specific methods (e.g., diagonalization) and plan a proof during the proof discovery process. E.g., the German mathematician Faltings, who proved Mordell's Conjecture, described in [14] that

> "We know from experience that certain inferences are usually successful under certain prerequisites. So first we ponder about a reasonable way to proceed to prove the theorem. In other words, we roughly plan: If we get a certain result the next result will follow and then the next etc. Afterwards we have to fill in the details, and to check whether the plan really works."

These insights make proof planning intriguing for interactive as well as for automated theorem proving.

A *computational* motivation for proof planning is the restriction of search which is necessary to cope with potentially infinite branching, long solutions, and numerous, even irrelevant, available axioms in theorem proving. Proof planning tries to avoid huge search spaces

1. by taking larger steps and
2. by employing global search control, as opposed to the more local search heuristics which are used for search control in automated theorem proving. That is, instead of making separate decisions at each choice point of proving at the (low) level of logical inferences, based on local clues, proof planning has some sense of the overall direction of the proof.

As for large steps, proof planning builds on tactics inherited from tactical theorem proving [16]. The operators in proof planning (called *methods*) have two aspects: On the one hand, they represent planning operators with declarative preconditions and effects. On the other hand, a method specifies a tactic, where a tactic executes a number of logical inferences. Thereby, larger chunks of proof steps are encapsulated in methods and thereby proof plans are an abstract representation of proofs.

The global control can be realized in several ways. In CL^AM, the step-case subproofs of inductive proofs are guided by the rippling heuristic [12, 21] which is well-suited for guiding proofs based on difference reduction. Below, we present control-rules and supermethods as other means of control in (extended) OMEGA.

2.1 Proof Planning in CL^AM

In order to enable a combination of tactical theorem proving with meta-level control, Bundy [8] introduced *methods* as (partial) specifications of tactics that specify in a meta-language the preconditions and effects of its application. In Figure 1 the structure of CL^AM's methods

```
name:          Prolog term
input:         sequent H==>G,
               H set of hypotheses, G goal
precondition:  list of conjuncts in
               meta-level language
postcondition: list of conjuncts in
               meta-level language
output:        list of sequents
tactic:        Prolog term
```

Figure 1: The method data structure in CL^AM.

is depicted. The methods serve as planning operators whose application yields the sequents from the output slot as subgoals.

The meta-level control came into play by (a) recognizing common proof plan patterns in families of proofs, for instance in proofs by mathematical induction or in diagonalization proofs, and by (b) discovering abstract goals and abstract heuristics that can guide the search for proofs. Proofs by mathematical induction reveal a common general structure displayed in Figure 2. This pattern is roughly to first find an appropriate induction schema and then

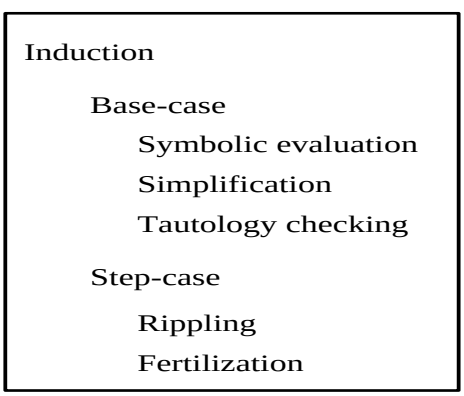

Figure 2: Structure of proofs by mathematical induction

to prove the conjecture for the base case, e.g., for $n = 0$, and for the step case, where the conjecture for a "successor" of the induction variable, e.g. of n, (called the induction conclusion) is proved provided the conjecture for the induction variable itself (which is called the induction hypothesis) holds. The step case pattern includes some kind of "fertilization", i.e., of applying the induction hypothesis to a rewritten induction conclusion such that a true formula results. This rewriting is subject to the abstract heuristic *rippling*. A meta-level goal in the step case is to reduce the differences between induction conclusion and induction hypothesis in order to enable a final fertilization. These differences are represented by

annotations, e.g., colours, to the induction conclusion. Axioms and definitions that belong to the initial state and which can be used to reduce the differences are annotated similarly.

The abstract search heuristic for proofs by mathematical induction, *rippling*, was introduced by Bundy [8] and Hutter [21]. It describes a systematic way to remove the differences, for example by moving the differences outward until the induction hypothesis can be applied to an inner part of the rewritten induction conclusion. For example, in proving the conjecture

$$\forall x, y, z.\ x + (y + z) = (x + y) + z \tag{2}$$

the induction hypothesis is

$$x + (y + z) = (x + y) + z \tag{3}$$

and the conclusion is

$$\boxed{s(\underline{x})} + (y + z) = (\boxed{s(\underline{x})} + y) + z \tag{4}$$

The boxes, excluding the underlined terms, denote the differences. The non-differences are called the *skeleton*. Rippling works by successively applying skeleton preserving definitions and axioms to the induction conclusion.

2.2 Original Proof Planning in OMEGA

In OMEGA, **domain knowledge** is stored in a hierarchically organized theory knowledge base. So far, proof planning in OMEGA employed axioms, definitions, and methods as the only domain knowledge. Theories may have parents they can inherit from. For instance, the theory *ordered-field* inherits, among others, from the theory *base* and a parent of the theory *limit* is *ordered-field*.

OMEGA's methods are frame-like structures specified in [20]. More specifically, these methods have the following slots: *declaration* of types, (annotated) *premises* and *conclusions, constraints, proof schema*, and *procedure*. From a logical (static) point of view, *premises* are sequents[4] from which the method logically derives the *conclusions*, and *conclusions* are sequents which the method is designed to prove. In `Refutation`, e.g., $(\Delta, \neg F \vdash \bot)$ is used to derive $(\Delta \vdash F)$.

The *constraints* are formulated in a meta-language and restrict the applicability of a method and the instantiations of parameters. If available, a *proof schema* is filled with a declarative schematic representation of a proof whose lines contain a label, a sequent, and a line-justification. This proof relies on the Natural Deduction (ND) calculus rules, on invoking tactics, or on invoking classical automated theorem provers such as OTTER [27]. Therefore, the line-justification consists of the name of an ND-rule, the name of a tactic or a prover, a variable, or OPEN in case the sequent is to be planned for. Additionally, the line-justification may include supporting lines. For instance, in `Refutation`

3. $\Delta \vdash F$ $\quad$ ($\neg$E;2)

states that the sequent $\Delta \vdash F$ is derived from the sequent in line 2 by the ND-rule $\neg$E. In what follows, names of lines abbreviate the corresponding sequents.

[4]Sequents $\mathsf{P} = (\Delta \vdash F)$, are pairs of a set Δ of formulas and a formula F in an object language that is extended by meta-variables for functions, relations, formulas, sets of formulas, and terms. The semantic of a sequent $(\Delta \vdash F)$ is that F can be inferred from Δ.

The program in the slot *procedure* is a special purpose procedure. The function *eval* that may occur in the *constraint* slot – see section 3.1 – returns *true* for an instantiation of variables, if its first argument is true before the procedure runs and if its third argument is true after the procedure's run. The second argument of *eval* is bound to the procedure's output.[5]

An example for a method is `Refutation`. Its annotations $\oplus, \ominus$ of *premises* and *conclusions* roughly indicate add ($\oplus$) and delete ($\ominus$) effects in STRIPS terminology, respectively. See a more detailed explanation below. The methods below omit some obvious details in order to make them more readable.

method: `Refutation`				
declaration	term: F			
premises	$\oplus$ L1			
conclusions	$\ominus$ L3			
constraints				
proof schema	L1. $\Delta, \neg F$	$\vdash$	$\bot$	(OPEN)
	L2. $\Delta,$	$\vdash$	$\neg\neg F$	($\neg$I,1)
	L3. Δ	$\vdash$	F	($\neg$E;2)
procedure				

The Planning Process in OMEGA

OMEGA's planning process searches the space of planning states, hence it is a state-space planning process as opposed to plan-space planning. A planning state contains a set of sequents represented by proof lines that is divided into open lines (goals) and closed lines (assumptions). Open lines are indicated by ? and closed lines by !. A proof planning problem is defined by an initial state specified by the proof assumptions) and the goal specified by the theorem to be proved. The planner searches for a solution, i.e., a sequence of methods that transforms the initial state into a state with no open lines.

Ignoring the $\oplus, \ominus$ annotations, *premises* and *conclusions* provide a purely logical description of a method. For proof planning, however, we need to know how it changes the planning state and how to introduce a method into the plan, e.g., by forward or backward planning. That is, it is important to specify which sequents can be used as inputs (**preconditions** in STRIPS notation) for a planning operator and which are outputs (**effects** with **add** and **delete** lists in STRIPS notation) of the method. Therefore, a translation of methods to a STRIPS-like representation of operators, used in OMEGA's planning, is derived from the annotated *premises* and *conclusions* of the method in the following way [34]:

- not annotated *premises* become !-lines in the **preconditions**,
- $\ominus$ annotated *premises* are translated to !-**preconditions** and to !-elements of the **delete** list of the operator,

[5] Binding meta-formulae (X $\leftarrow$ A) evaluate to true.

- $\oplus$ *premises* are introduced as ?-lines into the **add** list of the operator,
- not annotated *conclusions* become ?-**preconditions**,
- $\ominus$ *conclusions* are translated to ?-**preconditions** and introduced as ?-elements into the **delete** list of the operator,
- $\oplus$ *conclusions* are introduced as !-lines into the **add** list of the operator.

The operator corresponding to `Refutation`, for example, has the **precondition** ?L3, the **add-list** (?L1), and the **delete-list** (?L3). Hence, its input is ?L3 and its output ?L1. That is, `Refutation` is a backward operator (reducing a goal L3 to a subgoal L1). The following planning algorithm (in pseudo code) has been presented in [34] and will be extended in the sequel of this paper.

While there are ?-lines in the planning state

- Find all applicable instantiated operators:
  - Select an operator M,
  - find all matchings from M's **preconditions** with (? and !) elements of the current planning state. For each instantiation
  - evaluate the *constraints* of the instantiated M. This may further reduce the instantiations. An instantiated M is applicable, if the constraints evaluate to true for the instantiation.
- Select the best applicable instantiated operator M*. (backtracking point)
- Apply M* to the planning state by
  - inserting the sequents from the **add** slot of M* into the planning state,
  - deleting the sequents from the **delete** slot of M* from the planning state.

Successively expand methods of the complete plan to ND-subproofs.

The 'best' method M* is chosen according to a numeric *rating* of methods. Once a proof plan is found, all methods are successively expanded in order to obtain a calculus-level (ND) proof. For the expansion of methods, expansion functions are defined that are not discussed in this report. Sometimes, one expansion of methods yields plans that contain steps that are not ND-rules. Then these steps can be expanded further. The recursive expansion yields a hierarchical representation of a proof plan that is captured by the PDS (**P**lan **D**ata **S**tructure) data structure in OMEGA.

3 Our Extensions of OMEGA: Domain Knowledge and Architecture

Our experience with planning limit theorems and theorems from other domains [28], leads us to an extension of OMEGA's proof planner along the following dimensions. These extensions general rather than specific for OMEGA.

1. The **domain specific knowledge** to be employed for proof planning must include[6] (i) methods, (ii) control knowledge (control-rules), and (iii) constraint solvers. These extensions are discussed in sections 3.1, 3.2, and 3.3.

2. The **architecture** of OMEGA's planner must include the above knowledge sources and the planner has to be guided by a control unit that receives information from a monitor and that interpretes control-rules.

3. The planner should have multiple refinement strategies at its disposal (This is not discussed in this paper but in [28]).

The extended architecture of the proof planner is shown in Figure 3. Note the meaning of the different kinds of arrows in Figure 3: dashed arrows indicate choices, e.g., the control unit chooses a method. Solid black topped arrows indicate refinements, e.g., the application of a method refines the PDS. Solid white topped arrows indicate information delivery, e.g., the monitor delivers information to the control unit.

- The original planner architecture consists of a planning algorithm and a domain (theory) knowledge base that contains axioms, definitions, and methods. The application of methods can change the planning state.

- First, we add a control unit to the planner architecture and control-rules to the domain knowledge. The control unit interpretes control-rules in order to restrict the choices of methods and of goals before the planner decides about the applicability of methods, see § 3.3.

[6] apart from the obvious knowledge base with axioms, definitions, lemmata, previously proved theorems

While there are ?-lines in the planning state

- admissible methods := interpretation(control-rules, methods)
- for each admissible method M
  - find all matchings from M's **preconditions** with (? and !) elements of the current planning state. For each instantiation
  - evaluate the *constraints* of the instantiated M. An instantiated M is applicable, if the constraints evaluate to true for the instantiation.
- Select (best) applicable instantiated operator M*. (backtracking point)
- Apply M* to the planning state by
  - inserting the sequents from the **add** slot of M* into the planning state,
  - deleting the sequents from the **delete** slot of M* from the planning state.

Successively expand methods of the complete plan to ND-subproofs.

- The control works on the basis of information provided by a monitor that inspects the constraint state, resources, the planning state, and the planning history.
- Secondly, we add a constraint solver to the domain knowledge, see §3.2 for an explanation. Thereby we can take advantage of a lot of work performed in the CLP community. The constraint solver can be accessed via methods and can transform a constraint state. It provides answers to the application-conditions of methods too (about entailment or consistency of constraints).

  Now the domain knowledge (methods, control-rules, special constraint solvers) belongs to a theory in the theory knowledge base.
- Thirdly, the planning algorithm is changed such that it can invoke several refinement strategies rather than just one. Possible candidate strategies are: forward refinement, backward refinement, expansion, island planning, analogy, and refinement by the user [28]. In different ways, the different strategies introduce methods into the plan.

  The control unit supports the choice of refinement strategies in the planning process too. We do not discuss this extension here.

The flexible use of the expansion strategy turned out to be most interesting for planning limit theorems: Rather than always expanding methods when a complete plan is found or always expanding a supermethod instantaneously, the expansion strategy is picked flexibly according to the planning situation. This flexibility can be achieved by devising control knowledge that is evaluated according to the history, the current planning state, and the available resources.

The currently implemented expansion of supermethods as described in section 3.1.2 is less elaborate. Since the multistrategy-planning is not implemented yet, a program associated with supermethods expands a supermethod when it is introduced into the plan.

Multistrategy-Planning Algorithm `PLAN`(π)
Parameters: `sol` procedure for picking solutions

1. **Termination Check**: If `sol`(π) returns a solution, return it and terminate. If it returns *fail*, fail. Otherwise continue.
2. **Refinement**: Pick one of the following planning strategies that refine π:
   - forward-state-space
   - backward-state-space
   - expansion
   - analogical-driven proof plan construction
   - island-planning.
   - refinement by the user
3. **Recursive Invocation**: Call `PLAN` on the refined plan.

Table 1: Outline for multistrategy planning.

3.1 Methods for Planning Limit Theorems

The first examples of proof planning in OMEGA employed methods that were nothing else than (schematic) partial plans resulting from dividing (ND) proofs into subproofs. While this is fine for a first attempt such as [30], in general, methods may cover a much wider range than fixed ND-subproofs.

In this section, the relevant methods that belong to the *limit* domain theory or to its parent theories are presented. Note that `LimHeuristic` is the only method that is used exclusively for planning proofs for limit theorems. All other methods described below, are methods widely applicable at least for planning problems from ordered fields.

`LimHeuristic` is the central method in planning limit theorems. Foremost, the application of the other methods prepares the application of the `LimHeuristic` or prove inequalities. Depending on the particular problem, the `LimHeuristic` has to be applied a different numbers of times. For instance, in planning LIM+ it is applied once and for LIM* it is applied three times.

In the following, we consider basic methods and supermethods. We present them in a format that is still close to the actual implementation, in this technical report. Supermethod is a subclass of the method data structure that currently simulates planning with a restricted set of methods and control-rules. In the near future, the application of supermethods will, in a conceptually clean way, be associated with an expansion strategy of the planner.

3.1.1 Basic Methods

The *basic methods* that are employed for proof planning limit theorems are the methods Same, Same=, AndI, AndE, ImpI, ImpI, EquivE, EquivI, Skolem-b, Skolem-f, Mp-b, Backchain, IncreaseHyps, Focus, RemoveFocus, from the theory *base*, [7], the meth-

[7] Many of these correspond to ND-rules.

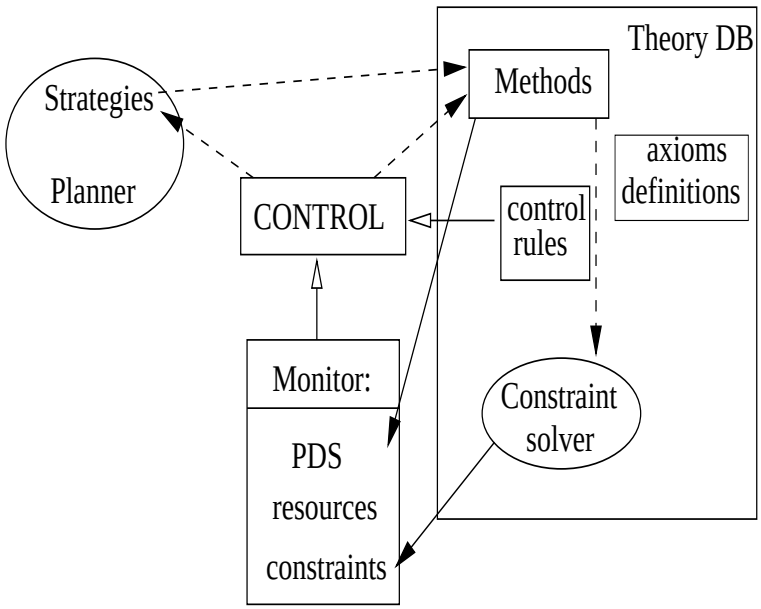

Figure 3: Extended architecture for proof planning.

ods `Solve`$_<$`b, Solve`$_<$`f, Solve`$_=$`b, Solve*, Solve*<b` from the theory *ordered-field*, and `LimHeuristic` from the theory *limit*. In the method names, the annotations `b` and `f` indicate the direction of the method application: **f**orward and **b**ackward, respectively. Backward methods are applied to goals whereas forward methods are applied to the assumptions of a planning state.

The `LimHeuristic` Method

The `LimHeuristic` method is a reconstruction of Bledsoe's limit heuristic in [6]. Similar to the mathematician's behavior described above, each application of `LimHeuristic` introduces a new auxiliary variable M upon which δ depends finally. M restricts the range of the object δ and its range is restricted by (1), (2), and (3) in turn. For instance, in planning LIM+, thm is $\Delta \vdash val(f(x_1) - l_1) < E_1 \rightarrow val(f(x) + g(x) - (l_1 + l_2)) < \epsilon$ and the subgoals are

1. $\Delta \vdash val(1) < M$
2. $val(f(X_1) - l_1) < E_1 \vdash val(f(x) - l_1) < div(\epsilon, 2 \cdot M)$
3. $\Delta \vdash val(g(x) - l_2) < div(\epsilon, 2)$

M is used to propagate range restrictions between the known constants and variables and δ as explained in section 3.2. `LimHeuristic` reduces a goal

$$val(a) < e_1 \rightarrow val(b) < \epsilon,$$

e.g., $val(f_1(x) - l_1) < E_1 \rightarrow val(f_1(x) + f_2(x) - (l_1 + l_2)) < \epsilon$, to three simpler subgoals (1), (2), and (3).

method: `LimHeuristic`	
declaration	term: $a, b, e_1, \epsilon k, l$ var: M
premises	$\oplus$(1),$\oplus$ (2), $\oplus$(3)
conclusions	$\ominus$thm, , $\oplus$(0)
constraints	eval($true, (k, l, \sigma), (k, l, \sigma) \neq \perp$)
proof schema	(0). (0) $\vdash$ $val(a) < e_1$ (HYP) (1). Δ $\vdash$ $val(k_\sigma) < M$ (OPEN) (2). (0) $\vdash$ $val(a_\sigma) < div(\epsilon, 2 * M)$ (OPEN) (3). Δ $\vdash$ $val(l_\sigma) < div(\epsilon, 2)$ (OPEN) l1. $\vdash$ $b = k_\sigma * a_\sigma + l_\sigma$ (CASextract) l11. Δ (0) $\vdash$ $val(b) < \epsilon$ (fix;l1,(1),(2),(3)) thmΔ $\vdash$ $val(a) < e_1 \rightarrow val(b) < \epsilon$ ($\rightarrow$I;l11)
procedure	`extract`(a, b)

The application condition reads as follows. If there exists (k, l, σ) such that `extract`(a, b)= (k, l, σ), then the method can be applied with the resulting instantiations of k, l, and σ.

The line-justification CASextract names a computer algebra tactic that can justify the equation $b = k_\sigma * a_\sigma + l_\sigma$, where k, l, and σ are computed by the oracle `extract` (see below) and $a_\sigma, k_\sigma, l_\sigma$ result from applications of σ to a, k, and l. During the expansion of `LimHeuristic`, CASextract produces a proof plan for this computation. The "fix" justification is an abbreviation for a fix subproof that proves the sequent in line l11 from the given support lines. '$\rightarrow$I' is the ND-rule implication introduction.

For planning purposes, the compact proof schema shown in the method is sufficient. For the interested reader, we give a more detailed proof schema. This level of detail illustrates why a fixed proof schema is possible for deriving thm from (1), (2), and (3).

NNo	S;D		Formula	Reason
(0).	(0)	$\vdash$	$val(a) < e_1$	(HYP)
(1).	Δ	$\vdash$	$val(k_\sigma) < M$	(OPEN)
(2).	(0)	$\vdash$	$val(a_\sigma) < div(\epsilon, 2 * M)$	(OPEN)
(3).	Δ	$\vdash$	$val(l_\sigma) < div(\epsilon, 2)$	(OPEN)
l2.		$\vdash$	$val(b) \leq val(k_\sigma * a_\sigma) + val(l_\sigma)$	(triang;l1)
l3.		$\vdash$	$val(b) \leq val(k_\sigma) * val(a_\sigma) + val(l_\sigma)$	(Mval;l2)
l4.	Δ	$\vdash$	$val(k_\sigma) * val(a_\sigma) + val(l_\sigma) \leq M * val(a_\sigma) + val(l_\sigma)$	(mult$\leq$;(1))
l5.	Δ	$\vdash$	$val(b) \leq M * val(a_\sigma) + val(l_\sigma)$	(trans$\leq$;l3,l4)
l6.	(0)	$\vdash$	$M * val(a_\sigma) < M * div(\epsilon, 2 * M)$	(mult<;(2))
l7.	(0)	$\vdash$	$M * val(a_\sigma) + val(l_\sigma) < M * div(\epsilon, 2 * M) + val(l_\sigma)$	(add<;l6)
l8.	Δ (0)	$\vdash$	$val(b) < M * div(\epsilon, 2 * M) + val(l_\sigma)$	(trans<;l5,l7)
l9.	Δ	$\vdash$	$M * div(\epsilon, 2 * M) + val(l_\sigma) < M * div(\epsilon, 2 * M) + div(\epsilon, 2)$	(add<;(3))
l10.	Δ (0)	$\vdash$	$val(b) < M * div(\epsilon, 2 * M) + div(\epsilon, 2)$	(trans<;l8,l9)
l11.	Δ (0)	$\vdash$	$val(b) < \epsilon$	(fix1;l10)
thm.	Δ	$\vdash$	$val(a) < e_1 \rightarrow val(b) < \epsilon$	($\rightarrow$I;l11)

How to read the more detailed proof schema?
'fix1' is an abbreviation for a fix arithmetic subproof that proves the sequent in line l11 from

the given support lines. 'triang' means the application of the triangle inequality, 'Mval' means the application of $A \leq val(B * C) + D \Rightarrow A \leq val(B) * val(C) + D$, mult$\leq$ means the application of $A < B \Rightarrow A * C + D \leq B * C + D$ for positive A, C, 'trans$\leq$' means the application of the transitivity of $\leq$, 'mult$<$ means the application of $A < B \Rightarrow A * C < B * C$ for positive C, 'add$<$' means the application of $A < B \Rightarrow A + C < B + C$.

Characteristically, the use of domain axioms and definitions belonging to the theory of $\mathbb{R}$ is encoded into the method rather than being visible at the proof planning level. Therefore, a search for applicability among the different axioms is not necessary.

How does the planner handle `LimHeuristic`?

- If a goal in the planning state <u>matches</u> **thm**, then `LimHeuristic`'s parameter, a, b, ϵ, e_1 are instantiated by the matcher. Then
- <u>*constraints* is evaluated</u>. That is, since the first argument of eval is *true*, the procedure `extract`(a, b) is invoked. The procedure `extract`(a, b) works as an oracle and outputs a list (k, l, σ), where k and l are terms and σ is a substitution, such that b can be represented as a linear combination of a, i.e., $b = k_\sigma * a_\sigma + l_\sigma$. The substitution σ computed by `extract` leaves b unchanged. For instance, if a is $g(X_2) - l_2$ and b is $(f(x) + g(x) - (l_1 + l_2))$, then `extract`$(a, b)$ returns the list $(1, (g(X_2) - l_2), [X_2/x])$.

  If `extract` runs successfully (i.e., result $\neq \perp$), then the method is applicable and $a_\sigma, k_\sigma, l_\sigma$ are bound to terms resulting from applications of σ to a, k, l.
- The goal, thm, can logically be inferred from this linear combination and from the subgoals (1), (2), and (3), as shown in the detailed proof schema. Therefore, the application of `LimHeuristic`, <u>removes the goal</u> and <u>introduces the new goals</u> (1), (2), and (3).

The `Solve` Methods

The methods `Solve`$_<$`b, Solve`$_=$`b, Solve*, Solve*<b, and Solve`$_<$`f` handle goals and assumptions, respectively, that involve linear equalities or inequalities. The methods `Solve`$_<$`b`, `Solve*<b`, and `Solve`$_<$`f` call the functions `tell`$_<$ or `ask`$_<$ that provide an interface between the methods and the constraint solver LINEQ, see section 3.2. `tell`$(a < b)$ returns $\perp$ if $(a < b)$ is inconsistent with the current constraint store. `ask`$(a < b)$ returns $\perp$ if $(a < b)$ is not entailed by the constraint store. In this case, the `Solve` method is not applied. Similar to the `Solve`$_<$ methods, we define a `Solve`$_=$ method that employs the function `tell`$_=$. The method's application can remove a goal $(a < b)$ from the state. The method `Solve`$_<$`b` has no preconditions, i.e., it produces no subgoals.

method: `Solve`$_<$`b`	
declaration	term: a, b var: $result$
premises	
conclusions	$\ominus$L1
constraints	eval($\sim$(occurs(a, b)), if var-in$(a < b)$ then $result \leftarrow$ `tell`$_<(a < b)$ else $result \leftarrow$ `ask`$_<(a < b)$, $(result \neq \perp)$)
proof schema	L1. $\Delta \quad \vdash \quad (a < b)$ (solverCS)
procedure	`tell`$_<(a < b)$, `ask`$_<(a < b)$

The method `Solve`$_<$`b` is applied to a goal $(a < b)$. Its application condition can be read as follows: In case the occurs-check for a, b fails (first argument of eval), the second argument of eval is evaluated: If a variable occurs in the goal $(a < b)$, then the function `tell`$_<$ is invoked that checks whether its argument $(a < b)$ is consistent with the current constraint store. Then LINEQ integrates $(a < b)$ into the constraint store and `tell`$_<$ returns *true* for *result*. Otherwise, *result* is $\perp$. If no variable occurs in $(a < b)$, then the function `ask`$_<$ is invoked with argument $(a < b)$. It checks whether the current constraint store entails $(a < b)$ already. If it does, then the result is *true*, otherwise $\perp$. If the output of the invoked function is not $\perp$ (third argument of eval), then the method is applicable.

In *proof schema*, the line-justification solverCS names a tactic that can recompute $(a < b)$ from the constraint store. During the expansion of Solve$_<$b, this tactic runs and provides a proof plan for its computation.

The method `Solve`$_<$`f` is applied to assumptions $(a < b)$, i.e., forward. It does not change the planning state, except for the constraint store. Since the assumption in L1 is not removed from the planning state, `Solve`$_<$`f` should be applied to an assumption only once. A second application would be redundant, see section 3.3.

method: `Solve`$_<$`f`	
declaration	term: a, b var: $result$
premises	L1
conclusions	
constraints	eval($\sim$(occurs(a, b)), $result$, $(result \neq \perp)$)
proof schema	L1. $\Delta \quad \vdash \quad a < b$ (solverCS)
procedure	`tell`$_<(a < b)$

The application condition of `Solve`$_<$`f` reads as follows. If the occurs check for a, b fails, then the function `tell` is invoked that checks whether $(a < b)$ is consistent with the current constraint store. If it is, `tell` returns *true*, otherwise $\perp$. The method is applicable if `tell` returns *true* only.

The next method, `Solve*b`, removes an inequality subgoal $(a' < c)$ (L3), if an assumption $(a < b)$ (L1) exists in the current planning state for which a and a' are unifyable. `Solve*b` returns a disjunctive subgoal (L2) and a subgoal (L4) which is a conjunction of equations derived from a substitution σ. For instance, if $\sigma = (a/x, b/y)$, then =conjunct is $(a = x \wedge b = y)$. =conjunct is produced by the procedure `subst-to-=conjunct`.

method: `Solve*b`	
declaration	term: a, a', b, c var: =conjunct
premises	L1 $\oplus$L2 $\oplus$ L4
conclusions	$\ominus$L3
constraints	`eval`$((\sigma =$`termmgu`$(aa'))\&$ subset$(\Delta 1, \Delta 2)$, =conjunct, $b_\sigma \leftarrow$ `substapply`$(\sigma b)\&\ c_\sigma \leftarrow$ `substapply`$(\sigma c))$
proof schema	L1. $\Delta 1$ $\vdash$ $(a < b)$ (j) L2. $\Delta 2$ $\vdash$ $(b_\sigma < c_\sigma) \vee (b_\sigma = c_\sigma)$ (OPEN) L3. $\Delta 2$ $\vdash$ $(a' < c)$ (arith;L1,L2,L4) L4. $\emptyset$ $\vdash$ =conjunct (OPEN)
procedure	`subst-to-=conjunct`(σ)

The application condition of `Solve*` reads as follows. If there is a most general unifier σ of a and a' and if $\Delta 1 \subseteq \Delta 2$, then the method is applicable with the particular instantiations of b_σ, c_σ, and =conjunct (the latter being the result of `subst-to-=conjunct`(σ)).

The 'arith' justification in line L3 of *proof schema* denotes a fix arithmetical subproof of $(a' < c)$ from $(a < b)$, $(b_\sigma < c_\sigma)$, and =conjunct.

A disjunctive subgoal $(a < b \vee a = b)$ can be removed by the following method `Solve*`$_<$`b` if it is applicable. `Solve*`$_<$`b` yields either no new subgoal or the subgoal $(a = b)$.

method: `Solve*`$_<$`b`	
declaration	term: a, b var: result, L
premises	$\oplus$ L
conclusions	$\ominus$L1
constraints	eval($\sim$occurs(a b), if var-in$(a < b)$ then *result* $\leftarrow$`tell`$_<(a < b)$ else *result* $\leftarrow$ `ask`$_<(a < b)$, if (result $\neq \perp$) then L $\leftarrow$ emptylist, else L $\leftarrow$ (L2))
proof schema	L1. Δ $\vdash$ $(a < b \vee a = b)$ (solverCS) L2. Δ $\vdash$ $(a = b)$ (OPEN)
procedure	`tell`$_<(a < b)$

The application conditions can be read as follows. When the occurs check on (a, b) failed and `tell`$_<(a < b)$ (`entail`$_<(a < b)$, respectively) succeeded, i.e., if the invoked function did

not return $\perp$, then `Solve*`$_{<}$`b` produces no subgoals. Otherwise, the method yields the new subgoal $(a = b)$. This is expressed by the third argument of eval in *constraint*that binds the meta-variable L to the empty list (no subgoals) or to L2 (subgoal $a = b$).

Methods from the Theory *base*

Some methods are applicable in all domains and therefore they belong to the theory *base*. Since most of them are well-known from the corresponding ND-rules, we describe only selected methods.

- `IncreaseHyp` enlarges the set of hypotheses $\Delta 0$ of an assumption. This is logically legal for any superset Δ of $\Delta 0$. However, one is not interested in arbitrary sets Δ. In the opposite, the infinite branching in search that is caused by almost arbitrary instantiations of Δ needs to be avoided, see section 3.3.

method: `IncreaseHyp`			
declaration	term: F		
premises	L1		
conclusions	$\oplus$ L2		
constraints	subset$(\Delta 0, \Delta)$		
proof schema	L1. $\Delta 0$ L2. Δ	$\vdash$ F $\vdash$ F	(j) (weaken L1)
procedure			

- `Backchain` is the ND-version of Bledsoe's "backchain" handling of assumptions [6].

method: `Backchain`			
declaration	term: F_1, F_2		
premises	$\ominus$L1, $\oplus$L2		
conclusions	$\oplus$L3		
constraints	subset(Δ_1, Δ)		
proof schema	L1. Δ_1 L2. Δ L3. Δ	$\vdash$ $F_1 \rightarrow F_2$ $\vdash$ F_1 $\vdash$ F_2	(j) (OPEN) ($\rightarrow$E;L1,L2)
procedure			

`Backchain` takes an implicational assumption $\Delta_1 \vdash F_1 \rightarrow F_2$ as input and yields a simpler assumption $\Delta \vdash F_2$ and an additional goal $\Delta \vdash F_1$ as output. Since `Backchain` may increase the search space unnecessarily, the choice of this method has to be controlled strictly.

- The method `Modus Ponens backward (Mp-b)` belongs to the *base* theory. `MP-b` takes an assumption $\Delta \vdash F_1$ and a goal $\Delta \vdash F_2$ and returns the new goal $\Delta \vdash F_1 \rightarrow F_2$ while removing $\Delta \vdash F_2$. Obviously, the application of `MP-b` has to be controlled strictly; otherwise it can produce new goals $F_{i1} \rightarrow (F_{i2} \rightarrow (\ldots F_2) \ldots)$ infinitely often.

method: `MP-b`	
declaration	term: F_1, F_2
premises	$\oplus$L1, L2
conclusions	$\ominus$L3
constraints	subset(Δ_1, Δ)
proof schema	L1. $\Delta \vdash F_1 \rightarrow F_2$ (OPEN) L2. $\Delta 1 \vdash F_1$ (j) L3. $\Delta \vdash F_2$ ($\rightarrow$E;l1,l2)
procedure	

- `Same=b` takes an assumption $(t_1 = t_2)$, a goal L3, and an assumption L1, where L1's formula results from L3's formula by replacing t_1 by t_2. The method removes the goal L3.

method: `Same=`	
declaration	term: F, F_1, t_1, t_2
premises	L1, L2
conclusions	$\ominus$L3
constraints	$F_1 =$ (termrploccs xFy) $\vee$ $F_1 =$ (termrploccs yFx) & subset$(\Delta 1, \Delta)$ & subset(Δ_2, Δ)
proof schema	L1. $\Delta 1 \vdash F$ (j1) L2. $\Delta 2 \vdash t_1 = t_2$ (j2) L3. $\Delta \vdash F_1$ (subst-apply;L1,L2)
procedure	

The application condition of `Same=b` reads as follows. If F_1 results from F by replacing t_1 by t_2 or vice versa and if Δ contains Δ_1 and Δ_2, then the method is applicable.

3.1.2 Supermethods

Hierarchical decomposition of methods is desirable in proof planning because restricting the search space by planning at a higher level without loosing the opportunity to expand it to a lower level plan can save search. Furthermore, a hierarchical presentation of the proof plan is easier to grasp by the user. The following decomposition techniques are known from the literature:

- Hierarchical task network planning (HTN) [37] that introduces abstract operators into a plan and then replaces the abstract operator by one of its reduction schemas. (The simplest form is planning with macro-operators [25] that represent fixed sequences of operators.) For proof planning, HTN planning cannot be used in exactly the same way because here the right decomposition may be computed from the planning situation rather than being one of the schemas fixed in advance.

- The proof planner *CLAM* [10], uses so-called supermethods that have a limited pool of operators they can invoke. For instance, the supermethod `step-case` has the submethods `ripple`, `fertilize`, and `base-case`. These, supermethods are expanded during the planning.

We want the decision when to expand a supermethod to depend on the planning situation, the available resources, and on the method itself. Hence, we extend the notion of supermethods. This extension takes into account that supermethods have two faces: one that exhibits the features of a method and another that amounts to, possibly complicated,[8] control knowledge. Therefore, our supermethods are methods that have premises and conclusions and at the same time provide control knowledge on how to build the expansion of the supermethod. Importantly, the control knowledge guiding the construction of the subplan can be specific for a particular supermethod because supermethods have the slot submethods and the slot control, as in the examples below.

How does the expansion of supermethods work currently?
A problem is created (containing one goal only). Then the planner is called with the set of methods given in the slot submethods. The sequence of methods in 'submethods' is the one in which the application of submethods is tried. Instead of the usual backtracking in planning, the supermethod's planning stops when no method is applicable. It returns the eventual resulting changes of the planning state. Supermethods for planning the limit-class theorems are:

name	submethods	control-rules
SOLVE-b	$\text{Solve}_{<}$b AndI	
SOLVE-f	$\text{Solve}_{<}$f AndE	`dismiss-a<b`
SOLVE*	Solve* AndI Solve=b $\text{Solve*}_{<}$b	
NORMAL	ImpI EquivI Skolem-b AndI	
UNWRAPHYP	Focus IncreaseHyp AndE Skolem-f Backchain	choose-focus increase-hyps attack-focus

`NORMAL` unpacks a goal similarly to simplification tactics . `NORMAL` is applicable if the input goal sequent's formula is not atomic.

[8] at least more complicated than the ordering of submethods as in *CLAM*.

supermethod: `NORMAL`	
character	unpredictable
declaration	meta-var: LIST term: Ψ
premises	$\oplus$LIST
conclusions	$\ominus$L1
appl-cond.	Ψ = formula(L1) & $\sim$ atom(Ψ)
submethods	ImpI EquivI Skolem-b AndI
control	()

`SOLVE-b` integrates unpacking and solving conjunctive inequality goals. Its applicability condition reads as follows. If $<$ occurs in the formula of L1, then try to apply the method.

supermethod: `SOLVE-b`	
character	unpredictable
declaration	meta-var: LIST term: F,
premises	$\oplus$LIST
conclusions	$\ominus$L1
appl-cond.	F= formula(L1) & $\sim$(termoccs($<$ F) = emptylist)
submethods	(Solve$_<$b AndI)
control	()

More interestingly, `Solve*` integrates all steps that are necessary to remove an inequality goal with the help of a similar assumption. The application condition says that a and a' from the assumption $\ldots \vdash a < b$ and from the goal $\ldots \vdash a' < c$ have to be unifyable.

supermethod: `SOLVE*`	
character	unpredictable
declaration	term: a,a',b,c meta-var: LIST
premises	L1, $\oplus$LIST
conclusions	$\ominus$L2
appl-cond.	formula(L1)= $a < b$ & formula(L2)= $a' < b$ & $\sigma \leftarrow$ termmgu($a'a$)
submethods	Solve* AndI Solve= Solve*$_<$b
control	()

Our most interesting supermethod is `UNWRAPHYP`. It decomposes an assumption such that a particular subformula S is extracted. In order to focus the attention to S, `Focus` has to

color S. The decomposition is continued until S is obtained, i.e., an assumption that is all colored. As opposed to 'unpredictable', the 'predictable' classification means that the main output can be anticipated without actually expanding the supermethod. For `UNWRAPHYP` this holds because the eventually resulting assumption is marked by a focus before applying `UNWRAPHYP`. However, the anticipation may not be reliable and in particular, it does not provide the subgoals LIST that arise during the full expansion.

supermethod: `UNWRAPHYP(ass, pos, goal)`	
character	predictable
declaration	term: F, meta-var: LIST
premises	$\ominus$L1
conclusions	$\oplus$LIST
appl-cond.	
submethods	(Focus, IncreaseHyp, AndE Skolem-f Backchain)
control	(choose-focus, increase-hyps, attack-latest)

In a multi-strategy planner, the expansion of supermethods can be one of the refinement strategies available to the planner. This expansion strategy yields the subplan that is introduced into the PDS at a hierarchically lower level, under the supermethod node. In case the invocation of refinement strategies is controlled by control-rules, the expansion can be invoked flexibly ,depending on the planning state and history as well as on resources and properties of the respective supermethod. For instance, the supermethod `UNWRAPHYP` does not have to be expanded immediately because a main output can be determined before actually expanding the method. However, an expansion could be preferred if enough resources are available and if the user wants to check the ND-proof. However, for some supermethods none of the resulting goals and assumptions can be predicted without actually expanding the method. In this case, an expansion will take place right away.

3.2 Combination of Proof Planning with Constraint Solving

In many mathematical proofs, logical steps are naturally combined with specialized reasoning such as computing integrals, solving polynomial equations, and finding instantiations of existentially quantified variables. For the construction of mathematical objects with certain properties, for example the existentially quantified δ in LIM+, pure proof planning can be difficult because infinitely many potential instantiations of the variable (the "object") may exist. A way to delay the instantiation (until the plan is completed) is the incremental restriction of the range of the variable by a domain-specific constraint solver that is combined with the planner.

Some of the advantages of integrating constraint solving are

- Logic and planning, respectively, provide very general data structures. Often, however, it is convenient to represent objects by specialized data types that can be efficiently handled, e.g., for rational and real numbers [18].

- For many constraints there exist very efficient specialized procedures for constraint solving (consistency check, entailment check, and simplification), e.g., for finite integer domains [17], for sets [36], etc.

3.2.1 General Framework for Constraint Solving

Jaffar and Maher suggest an abstract procedural semantics for constraint solvers in [22]. We briefly and simplified[9] review this framework and then we present below our self-made constraint solver LINEQ that implements the necessary operations.

A constraint solver acts upon a constraint state, (C, S), where C and S are multisets of constraints. C is the set of so-called active constraints (constraint store) and S is the set of so-called passive constraints. There is one other state denoted by *fail*. Constraint solving is modeled by a transition system that is parametrized by a predicate `consistent` and a function `infer`. Three transitions of the constraint state are $\rightarrow_c$, $\rightarrow_i$, $\rightarrow_s$,

$$(C, S) \rightarrow_c (C, S \cup c)$$

if c is a constraint told to the constraint solver,

$$(C, S) \rightarrow_i (C', S')$$

if $(C', S') = \texttt{infer}(C, S)$.

$$(C, S) \rightarrow_s (C, S)$$

if $\texttt{consistent}(C)$ or else

$$(C, S) \rightarrow_s fail$$

The $\rightarrow_c$ transitions introduce constraints into the constraint solver, $\rightarrow_s$ transitions test whether the constraint store is consistent, and $\rightarrow_i$ transitions infer more active constraints (and possibly modify the passive ones).

A `tell` c constraint is handled by $\rightarrow_{cis}$ transitions; and an `ask` constraint c is handled by $\rightarrow_{ci}$ transitions with $(C, S) \rightarrow_i (C, S - \{c\})$. `ask` constraints are tested for entailment from C only. For instance, in planning LIM+, the `ask` constraint $(0 < \delta_1)$ is entailed by $(0 < D)$ and $(D < d_1)$ from the constraint store.

In order to implement the abstract operational model of constraint solvers, several operations have to be implemented. These include: a **satisfyability test** to implement `consistent` and `infer` and the **projection** of the constraint state onto a set of variables to compute an *answer* constraint from the final store. For the extraction of *answer*, the problem at hand is to obtain a useful representation of the projection of constraints C w.r.t. a given set of variables. Usability usually means conciseness and readability [22].

Most constraint solving systems represent constraints in C in a solved form which is a convenient representation of the projection of the solution space with respect to any set of variables. Usually, solved form means a format in which satisfyability of C is evident.

[9] We do not consider constraint states that include formulae that are not constraints because we do not combine logical and constraint inferences in one calculus.

3.2.2 Combining Proof Planning with Constraint Solving

In proof planning, the constraint-handling component serves two main purposes: Firstly, it is used during the process of proof planning to determine whether a `Solve` method can be legally applied. This is realized by checking consistency or entailment of a constraint with the constraint store. Secondly, after the completion of the proof plan, the constraint store is condensed into an answer assertion *answer* about the values of variables. *answer* is the starting assumption for the final textbook style proof as discussed in section 5. It justifies inequalities and equalities that follow from the final constraint store.

Interface

An interface between proof planning and constraint solving can be provided by methods. For proof planning limit theorems, the most important methods at the interface between the proof planning and constraint-handling component are $\texttt{Solve}_{<}\texttt{b}$, $\texttt{Solve}_{<}\texttt{f}$, `Solve=b`, and $\texttt{Solve*}_{<}\texttt{b}$. Their purpose is to remove an equational or an inequality goal by adding it to the constraint state. Each of the `Solve` methods is applicable only if the function called returns *true*. In this case, it will later be possible to derive the proved inequality from the final constraint store.

The functions `tell` and `ask`, respectively, are invoked by the `Solve` methods and access the constraint solver. Which function to call depends on the kind of constraint. $\texttt{tell}(c)$ checks the consistency of c with the current constraint store C and may add the (in)equality to C, whereas $\texttt{ask}(c)$ checks whether (c) is entailed by C.

Through the *application conditions* $\texttt{Solve}_{<}\texttt{b}$,$\texttt{Solve}_{=}\texttt{b}$, and `Solve*b` determine whether a goal $(a < b)$ or $(a = b)$, respectively, is a `tell`-constraint or `ask`-constraint. The access of the constraint solver via `tell` is chosen when the inequality at hand contains a (existentially quantified) variable that can be solved for. Otherwise, `ask` is chosen. The reason is that an (in)equality **goal** $(a < b)$ that contains constants and universally quantified variables only, e.g. $0 < \delta_1$, cannot be introduced into the constraint store without a loss of generality of the proof, whereas implicitly existentially quantified variables can. For an assumption the situation is different, of course. It can be introduced into the constraint store in any case. For instance, for an assumption like $\forall\delta_1(\delta_1 > 0 \rightarrow \ldots)$, δ_1 is universally quantified; its restriction $(\delta_1 > 0)$ is assumed as a hypothesis

3.2.3 The Constraint Solver LINEQ

For the first experiments we used our own constraint solver LINEQ that is capable of handling value constraints that are expressed by linear equalities and inequalities over $\mathbb{R}$ that may contain terms $val(x)$ for a variable or constant x. LINEQ was motivated by a particular application, so the general-purpose domain $\mathcal{R}_{Lin}$[10] has been extended by the ad-hoc addition of the absolute value function val and an internal handling of this interpreted function. For instance, $val(t) < 0$ is trivially invalid.

For `tell`-constraints LINEQ checks the consistency of an input with the constraint store and propagates value restrictions. In the process of planning a proof, the constraint state will be modified with every application of $\texttt{Solve}_{<}\texttt{b}$ or similar methods. Every time a new inequality is passed to LINEQ, it is necessary to immediately decide the consistency of the

[10] $\mathcal{R}_{Lin}$ denotes the domain of linear arithmetic over the real numbers with the function symbol $+$ and the predicate symbols $=, <, \leq$.

original store with the new inequality. This situation calls for a representation of the constraint store by a solved form in which every constraint of a variable that can possibly be propagated is stored explicitly. The explicit representation of the constraint stores also facilitates backtracking to earlier PDS nodes (or planning states) by storing complete constraint stores in nodes.

Constraint Representation

In order to be able to handle equalities as well as inequalities, we consider "equality classes" of terms that have been claimed to be equal at some point in the planning process by adding a respective equality. The constraint store is represented as a (disjunctive) list of branches, each of which contains (for every known equality class) lists of terms that denote upper and lower bounds. This representation allows for case splits: For example, the solution of the inequality $a \cdot x < 1$ for x has two branches, one where a is negative and one where it is positive. In such a case, the branch of the constraint store that has been worked on would be split in two, each of them containing one of the solutions.

Every time a new `tell`-constraint (equality or inequality) is added, the constraints is propagated until no new information can be extracted. This is necessary for a consistency check, anyway, and ensures that the constraint store is always represented explicitly.

Some Functions of LINEQ

LINEQ has functions to initialize the constraint state, to check the (in)consistency of a constraint store, to solve a linear (in)equality for a variable, to propagate constraints, and to project a constraint store onto a set of variables.

The constraint store component of the first planning state is initialized with { `[ ]` }, i.e., the constraint store that only contains the empty branch.

The `add-inequality` and `add-equality` functions tell new information to the store and propagate it. Applying the $\texttt{Solve}_{<}\texttt{b}$ method in a planning state calls the `add-inequality'` function, passing as arguments the old constraint store and an inequality. If these are consistent with each other, the function returns a new constraint store that contains all information that could be propagated from them. The $\texttt{Solve}_{=}\texttt{b}$ method uses the `add-equality'` function in the same manner. The algorithm to add an inequality to the constraint store is described in the following pseudocode.

Adding an inequality `a < b` *to a branch* `B`*:*

```
add-inequality(B, a < b, H):
      IF a < b is trivially invalid THEN RETURN { }
 ELSE IF a < b is trivially valid THEN RETURN { B }
 ELSE IF a < b contains no variables THEN RETURN { }
 ELSE IF a < b appears in the history H THEN RETURN { }
 ELSE
  R := \emptyset
  FOR every variable x in a and b DO
   Solve a < b for x
   FOR every possible solution DO
    B' := B
    B'' := \emptyset
    IF the solution yields an upper bound u for x THEN
     IF the upper bound is not already entailed by the branch THEN
      Upper(x) := Upper(x) \cup {u} (in B')
      S' := { B' }
      FOR every lower bound l of x DO
       S' := add-inequality'(S', l < u, H \cup { a < b })
      FOR every side condition C of the solution DO
       S' := add-inequality'(S', C, H \cup { a < b })
    IF the solution yields a lower bound l for x THEN
     (analogously)
    R := R \cup B''
  RETURN R
```

Adding an equality `a = b` *to a branch* `B`*:*

```
add-equality(B, a = b):
      IF a = b is trivially invalid THEN RETURN { }
 ELSE IF a = b is trivially valid THEN RETURN { B }
 ELSE IF a = b contains no variables THEN RETURN { }
 ELSE
  FOR every variable x in a and b DO
   RHS := Solve=(a = b, x)
   Ex := equality class of x in B
   IF RHS doesn't appear in Ex THEN
    IF RHS is in the equality class of another variable THEN
     ; merge the ECs
     Er := equality class of RHS in B
     B' := remove Er from B
     FOR every r in Er DO
      B' := add-equality(B', x = r)
     S' := { B' }
     FOR every upper bound u of Ex and every lower bound l of Er DO
      S' := add-inequality'(S', l < u)
     FOR every lower bound l of Ex and every upper bound u of Er DO
      S' := add-inequality'(S', l < u)
     RETURN S'
    ELSE
     B' := add r to the equality list of x
     S' := { B' }
     FOR every member t of Ex except for x and t DO
      S' := add-equality'(S', r = t)
     FOR every upper border u of Ex DO
      S' := add-inequality'(S', r < u)
     FOR every lower border l of Ex DO
      S' := add-inequality'(S', l < r)
     RETURN S'
```

The `add-inequality` function takes as arguments an inequality and a branch of a constraint store; it returns a list of branches that describe all possible situations satisfying both the old branch and the new inequality. The `add-inequality'` function is its extension that takes an entire list of branches and executes `add-inequality` on each branch.

First, some conditions about the consistency or inconsistency of the inequality are tested[11]; moreover, we test if the inequality appears in a history list `H` to avoid loops. Then the inequality is solved for each variable that it contains. A solution of a linear inequality is an upper or lower bound for the variable along with a number of side conditions. (Side conditions discriminate among different branches of the solution. In the example of $a \cdot x < 0$ above, one solution would have a side condition of $a < 0$ and a lower bound of 0 for x, the other would have a side condition of $a > 0$ and an upper bound of 0 for x.) If this bound is not already known (if, for example, we obtain an upper bound that is a number greater

[11] Of course, we can only employ a number of heuristics to test for obvious situations; general validity of arithmetic inequalities is undecidable.

than another number that already appears in the list of upper bounds), it is inserted into the appropriate bound list for the variable and compared against the bounds of the other side. This comparison produces several new inequalities that are recursively added to the constraint store. Afterwards, all side conditions are added to the store. These steps can produce several new branches, one for every branch of the inequality's solution. A branch can be closed, i.e. removed from the constraint store, if it contains an inconsistency (for example, inequalities of the form $x < x$ or $0 < -1$ are trivially invalid). A constraint store is inconsistent if it contains no more open branches.

The `add-equality` function, which takes as argument an equality and a branch and returns a list of branches, is extended to the function `add-equality'` operating on lists of branches in the same manner as `add-inequality` is extended to `add-inequality'`. After some preliminary tests for obvious validity and invalidity of the argument equality, it is solved for every variable x it contains. This time, a solution is simply a term, called RHS in the pseudocode. There are three possible cases for RHS.

1. RHS is already known to be equal to x, i.e. it is a member of its equality list. Then we are already done.

2. RHS is in the equality list of any other variable. Then we successively add every member of RHS's equality list to x's equality list. Every time we add a new member, we obtain a number of new equalities (to the former members of the equality list) and inequalities (to the upper and lower bounds of x), which are recursively added to the store. We also recursively add all inequalities of the form $l < u$, where l is a lower bound of either x or RHS and u is an upper bound of the other class.

3. RHS isn't in the equality list of any other variable. Then we add it to the equality list of x, obtaining a number of new equalities (to the former members of the equality list) and inequalities (to the upper and lower bounds), just like in the previous case.

The Projection Function

In constraint solvers, the projection of a constraint c_0 onto variables $\overline{x}$ to obtain a constraint c_1 such that $\mathbb{R} \models c_1 \leftrightarrow \exists \overline{x}.c_0$. The projection aims at computing the simplest c_1 with fewest quantifiers. (In general it is not possible to eliminate all uses of the existential quantifier.)

In a ND-proof that can be produced from the proof plan, every goal that was removed by a `Solve` method by planning must be logically justified. As we said above, we achieve this by at the beginning of the proof asserting the formula *answer* that is a projection of the final constraint store w.r.t. the (implicitly existentially quantified) variables. From this formula, every line that was justified by adding an (in)equality to the constraint store follows by means of logic and arithmetic. This formula *answer* can be extracted from the final constraint store in a very straightforward way: Without loss of generality, choose on of the branches of the constraint store and transform it into a conjunction of (in)equalities.

Example

Let us consider an example. In the proof of the LIM+ theorem, the following constraints are told to the constraint solver in a row: $0 < D, 1 < M, E_1 < div(\epsilon, 2 * M), x = X_1, E_2 < div(\epsilon, 2), x = X_2, D < \delta_2, D < \delta_1$. From the constraints $1 < M$ and $E_1 < div(\epsilon, 2 * M)$, for instance, the new upper bound $div(\epsilon, 2)$ of E_1 is propagated.

This leads to several applications of the `add-inequality'` and `add-equality'` functions and to a final constraint store that looks as follows:

$0 < E_2 < div(\epsilon, 2);$
$0 < D < \delta_2, \delta_1;$
$0 < E_1 < div(\epsilon, (2 * M)), div(\epsilon, 2);$
$1 < M < div(\epsilon, (2 * E_1))$
$-\infty < X_1 = x = X_2 < +\infty$

That is, a lower bound for E_2 is 0 and an upper bound for E_2 is $1/2 * \epsilon$; a lower bound for D is 0 and an upper bounds are δ_1, δ_2, etc. The above constraint store contains a single branch that contains information about five equality classes, most of which only contain a single variable; x, x_1, and x_2, however, have been collected in one equality class.

Suppose we wanted to introduce another inequality $M < E_1$: Both variables are existentially quantified, so this would add E_1 to the list of upper bounds of M and recursively add the inequality $1 < E_1$ (which, in turn, will produce inequalities $1 < u$ for every upper bound u of E_1). Likewise, we would have to add M to the list of lower bounds of E_1 and recursively add, for every upper bound u of E_1, the inequality $M < u$.

From the final constraint store of the LIM+ plan, our current projection algorithm that is similar to Bledsoe's SupInf, computes the *answer* w.r.t. E_1, E_2, D
$0 < E_2 \wedge E_2 < \frac{\epsilon}{2} \wedge 0 < D \wedge D < \delta_2 \wedge D < \delta_1 \wedge 0 < E_1 \wedge E_1 < \frac{\epsilon}{2} \wedge$
which contains the core assumption a mathematician would make in the proof, namely, "Let $\epsilon_1 < \frac{\epsilon}{2}, \epsilon_2 < \frac{\epsilon}{2}, \delta < \min\{\delta_2, \delta_1\}$." Sometimes an *answer* contains conjuncts that are not relevant for the proof.

Using a more General Constraint Solver

Actually, we could have taken off-the-shelf a constraint solver for linear arithmetic over the real numbers, e.g., CLP($\mathcal{R}$) [23]. In this case we would have to take care of so-called non-basic constraints containing the interpreted absolute value function *val*. This can be done by employing lemmata about the interpreted (or user-defined) function, e.g., $0 \leq x \rightarrow |x| = x$, as guarded constraints [33] or as user-defined propagators with a as done in Oz [31]. (BTW, This is similar to the augmentation module known from the integration of Linear Arithmetic into Nqthm.)

In our proof planning framework, the introduction/employment of a lemma can be achieved in the following way: If c is a non-basic constraint that cannot be solved because it contains an interpreted function term t, then the constraint solver puts c into the S-part (passive constraints) of the constraint state (C, S). c is activated only if a lemma exists in the planning state that can be used to rewrite c into a basic constraint. If the lemma can be used only to rewrite c into a non-basic constraint c', then c is replaced in S by c'. A control-rule has to be devised that looks at S and, in the simplest case, selects `Solve`$_<$`f` or `Solve`$_=$`f` with an instantiation of its parameter whose lhs or rhs matches t. By applying the respective method, an (in)equality is told to the constraint solver that helps to solve the constraint c. That is, as opposed to constraint solvers, the control lives outside and is not predefined for each lemma separately.

Suppose, for example, $(val(U) < M)$ is a passive constraint, the assumption $(0 \leq X \rightarrow X = val(X))$ is in the planning state, and $(0 \leq U) \in C$. Then the constraint $(U = val(U))$ is told to the constraint solver and $(val(U) < M)$ becomes activated, which results in the

basic constraint ($U < M$) in C.

3.3 Mathematical Control Knowledge

In planning, several decisions have to be made. There are choices between alternatives for strategies, methods, goals to work on, and instantiations of variables. Control knowledge in proof planning is devised to reduce the alternatives and to prefer certain user-friendly proof plans. E.g., having certain steps belonging together as neighboring steps in a sequence may be better comprehensible for a user.

Numerous experiences indicate the superiority of a separate representation of control knowledge by control-rules for the user's comprehension, for experiments with different rules, for modifications, and for learning control knowledge. This modular and declarative representation of control-rules, that is similar to a small expert system, has been useful, provided the interpretation works efficiently. For efficient algorithms see, e.g., the Rete-Algorithm in OPS5 or, more recently [13]. For a summary and evaluation, see [40].

In this report, we describe a control-rule mechanism. Our control-rules contain decidable meta-predicates, encoded in LISP functions, that inspect the current planning state, the planning history, resources, and the current partial proof plan.

3.3.1 Syntax of Control-Rules

Currently, we distinguish the following classes (kinds) of control-rules that correspond to the different decisions of the planner.

- `strategy`
- `method`
- `sequent`, with the subclasses `goal` and `assumption`,

Other kinds, such as `abstraction` are conceivable and planned as well as rules with an extended syntax that allows for a more complicated control structure.

The syntax of control-rules in (extended) OMEGA is defined as

```
Control-rule :=  (control-rule <Name>
                               (kind <Kind>)
                               (if <If-part>)
                               (then
                                   (<To-do-part>)
                                   (side-effect <Side-effect-part>)))
```

The syntactic definitions are as follows

```
<Name>                 := lisp-symbol
<Kind>                 := method | sequent | binding
<If-part>              := <Meta-predicate> | (and <If-part> <If-part>)
<Meta-predicate>      := LISP expression
<To-do-part>           := () | <Choice> |
```

```
                              (and <To-do-part> <To-do-part>)
<Choice>                   := select <Alternative-list> |
                              reject <Alternative-list> |
                              prefer <Alternative-list> |
                              iterate <Alternative-list>|
<Alternative-list>         := list of strategies | list of methods |
                              list of sequents | list of bindings
<Side-effect-part>         := () | conjunction of side-effects
```

Meta-predicates return all satisfying binding alternatives in case an argument is not instantiated; otherwise they return a truth value. Examples for meta-predicates are `current-goal`(x) and `last-method`(x) which yield instantiations for x, if x is a variable. Otherwise they return the truth value, e.g., of `current-goal`(a) for the instantiation (a/x). Meta-predicates can be written by the user but some frequent ones are available in (extended) OMEGA.

`Choice` refers to different strengths of the restriction. select and reject are stronger than prefer. Note that select and reject rules can result in an incompleteness of the planner. The `iterate` mode is an extended `prefer` mode. It allows for sequencing several methods by means of a control-rule. This mode can avoid a repeated expensive evaluation of the if-part of control-rules. Side-effects are LISP functions that can set flags etc.

Interpretation of Control-Rules

Before matching the preconditions of operators with lines of the current state, the planner presents a list of alternatives of a certain kind to the control-rule interpreter via an interface. The interpreter returns a (reduced) list of alternatives via the interface. That is, the matching and the search effort decreases.

Given a list of alternatives of kind k, the interpreter processes the control-rules of kind k in the sequence in which they are stored. The if-parts are evaluated as long as no conjunct evaluates to false for at least one instantiation. `select`(*list*) returns an (ordered) list, *list*, of alternatives which is then intersected with the current list of alternatives. `prefer`(x) reorders the list of alternatives such that x is the first element if it was in the list before. `reject`(*list*) removes *list* from the current list of alternatives. `iterate`(*list*) determines the next method choices in the planning process according to *list*.

3.3.2 Control-Rules for Planning Limit Theorems

In the following, we present a set of control-rules that produced a satisfying search behavior in planning limit theorems as documented in section 6. With the given control-rules, we want to demonstrate the line of reasoning for designing control-rules rather than competing for the most efficient control. Most of the following rules are designed in order to capture the following **global control story**:
First the goal is normalized. Linear inequality goals can be satisfied by `Solve`$_<$`b`, `SOLVE*`, or by `LimHeuristic`. The latter requires some preparation by `UNWRAPHYP` and `MP-b`. In planning the limit theorems, `UNWRAPHYP` extracts a particular subformula s from an assumption in order to afterwards employ s as an antecedent in a goal that can be reduced by `LimHeuristic`. In order to guide the unwrapping of s of an assumption, `Focus` annotates s

and thereby puts a local focus on this subformula. The purpose of this focus is to concentrate operator applications on extracting the colored formula.

Now the control-rules together with some explanation of their intended purpose follow.

```
(control-rule attack-goal
        (kind method)
        (if (and (goal-matches ("goal" (less (val "x") "y")))
                 (most-similar-subterm-in ("goal"
                                           "ass"
                                           "pos"))))
         (then
             (prefer ((Solve<b "goal")
                      (SOLVE* "goal")
                      (UNWRAPHyp () ("ass") ("goal"))))))
```

The intention of `attack-goal` is an attempt to apply $\texttt{Solve}_{<}\texttt{b}$ or alternatively `SOLVE*` if a goal is of the form $val(x) < y$, and if these two fail to be applicable, to move on to the preparations for `LimHeuristic` by `UNWRAPHYP`.

The evaluation of the meta-predicate `most-similar-subterm-in` returns for each previously resulting instantiation of `"goal"`, `most-similar-subterm-in` returns instantiations of `"ass"` and `"pos"`. The meta-predicate computes the formula s at position `pos` in some assumption `ass` such that s is most similar to the goal. The similarity is measured by function symbol occurrences and characterizes the suitability of antecedents for `LimHeuristic` to succeed. The evaluation of `most-similar-subterm-in` is expensive. A repeated evaluation of this meta-predicate can be avoided by designing a different set of control-rules including one that iterates over `Focus, Increase-Hyp` and `UNWRAPHYP`.

After `UNWARPHYP` stopped and returns an assumption that is all colored (unwrapped), the unwrapping business is completed by removing the color altogether because it is no longer necessary. This is expressed in the rule `after-UNWRAPHYP` (We did not integrate `RemoveFocus` into `UNWARPHYP` because `RemoveFocus` is applied to a particular assumption, the unwrapped one, and hence determines *the* assumption, `"ass"`, that is one argument for the then following `Mp-b` method.)

```
(control-rule after-UNWRAPHYP
              (kind method)
              (if (and (last-method UNWARPHYP)
                       (and (latest-assumption "ass")
                            (unwrapped-focus "ass"))))
              (then
               (select ((RemoveFocus () ("ass"))))))
```

After the unwrapping extracted an assumption *focus(F)* and the color removal provided the uncolored assumption F, `SOLVE*` or alternatively `Mp-b` – the last preparation for `LimHeuristic` – are tried to be applied as formulated in the next control-rule. `SOLVE*` is tried first because it produces simple or no subgoals as opposed to `LimHeuristic` preceeded by `Mp-b`.

```
(control-rule attack-unwrapped
              (kind method)
              (if (and (last-method RemoveFocus "ass")
                            (last-UNWRAP "ass" "goal"))))
              (then
               (select ((SOLVE* "goal" ())
                        (Mp-b "goal" ("ass"))))))
```

`Mp-b` is exclusively applied in order to provide a goal that `LimHeuristic` can handle afterwards.

```
(control-rule choose-LimHeuristic
              (kind method)
              (if (last-method Mp-b "goal"))
              (then
                (prefer ((LimHeuristic "goal" ()))))))
```

The next rule can be read as: `Solve`$_<$`f` should be applied to an assumption only once because a second application is redundant.

```
(control-rule dismiss-a<b
                (kind sequent)
                (if (and (already-applied (Solve<f "ass"))
                (then
                (reject-sequents (() ("ass") ))))
```

Finally, we have control-rules that belong to the control of supermethods. For instance, the three following control-rules govern the planning in the expansion of supermethod `UNWRAPHYP`. First `Focus` is choosen with paramters. `choose-focus` selects the subformula of an assumption to focus on. Then `IncreaseHyp` is preferred in order to adjust the hypotheses set of the picked assumption sequent to the hypotheses set of the goal sequent. Finally, only assumptions that were derived latest and that carry a focus are admitted as an input of the next `UNWRAPHYP` submethods. `attack-latest`

```
control-rule choose-focus
       (kind method)
       (if (and (no-focus)
              (most-similar-subterm-in ("goal"
                                        "ass"
                                        "pos"))))
       (then
          (select ((Focus () ("ass") ("pos"))))))

(control-rule increase-hyps
        (kind method)
        (if (and (last-method Focus)
```

```
                (and (latest-assumption "ass")
                     (and (current-goal "goal")
                          (hyps-of "goal" "hyps")))))
        (then
           (prefer ((IncreaseHyp () ("ass") ("hyps"))))))

(control-rule attack-latest
      (kind sequent)
      (if (sub-of-latest-assumption(focus "ass"))
      (then
       (select-sequents (()("ass")))))
```

4 Exemplary Proof Planning for LIM+

LIM+ is a limit theorems with a relative simple proofs. Since an exemplary detailed description should be comprehensible and not too boring, we present LIM+ rather than, say, LIM*. A proof plan for the more difficult proof of LIM* is provided, however, in Figure 4.

Planning for LIM+ starts with the goal
$\emptyset \vdash \forall\epsilon_1\exists\delta_1\forall x_1(0 < \epsilon_1 \to 0 < \delta_1 \wedge val(x_1 - a) < \delta_1 \to val(f(x_1) - l_1) < \epsilon_1) \wedge$
$\forall\epsilon_2\exists\delta_2\forall x_2(0 < \epsilon_2 \to 0 < \delta_2 \wedge val(x_2 - a) < \delta_2 \to val(f(x_2) - l_1) < \epsilon_2)$
$\to \forall\epsilon\exists\delta\forall x(0 < \epsilon \to 0 < \delta \wedge val(x - a) < \delta \to val((f(x) + g(x)) - (l_1 + l_2)) < \epsilon)$ After all the skolemizations, we use the Prolog notation: ϵ is a constant, δ_1, δ_2, x are Skolem functions, and E_1, E_2, D, X_1, X_2 are variables.

1. The `NORMAL` application consists of

   (a) `ImpI` results in:
   $\forall\epsilon_1\exists\delta_1\forall x_1(0 < \epsilon_1 \to 0 < \delta_1 \wedge val(x_1 - a) < \delta_1 \to val(f(x_1) - l_1) < \epsilon_1) \wedge$
   $\forall\epsilon_2\exists\delta_2\forall x_2(0 < \epsilon_2 \to 0 < \delta_2 \wedge val(x_2 - a) < \delta_2 \to val(f(x_2) - l_1) < \epsilon_2)$
   $\vdash \forall\epsilon\exists\delta\forall x(0 < \epsilon \to 0 < \delta \wedge val(x - a) < \delta \to val((f(x) + g(x)) - (l_1 + l_2)) < \epsilon)$
   and the new assumption $H \vdash H$ for H:
   $$\boxed{\begin{array}{l}\forall\epsilon_1\exists\delta_1\forall x_1(0 < \epsilon_1 \to 0 < \delta_1 \wedge val(x_1 - a) < \delta_1 \to val(f(x_1) - l_1) < \epsilon_1) \wedge \\ \forall\epsilon_2\exists\delta_2\forall x_2(0 < \epsilon_2 \to 0 < \delta_2 \wedge val(x_2 - a) < \delta_2 \to val(g(x_2) - l_2) < \epsilon_2).\end{array}}$$

   (b) `Skolem-b` is applied to the goal, giving the subgoal $H \vdash 0 < \epsilon \to 0 < D(\epsilon) \wedge val(x(D) - a) < D(\epsilon) \to val((f(x(D)) + g(x(D))) - (l_1 + l_2)) < \epsilon)$.
   In the following, we abbreviate by omitting the Skolem function arguments. After all the skolemizations, we use the Prolog notation: ϵ is a constant, δ_1, δ_2, x are Skolem functions, and E_1, E_2, D, X_1, X_2 are variables.

   (c) `ImpI, ImpI` is applied to the current goal. It moves the antecedent $(0 < \epsilon)$ and then $(0 < D \wedge val(x - a) < D)$ from the conclusion to the hypotheses giving a subgoal with the resulting conclusion
   $$\emptyset \vdash val((f(x) + g(x)) - (l_1 + l_2)) < \epsilon$$
   and with the set of hypotheses
   $\Delta = \boxed{\{H, (0 < \epsilon), (0 < D \wedge val(x - a) < D)\}}$
   and giving the new assumptions

i. $(0 < \epsilon) \vdash (0 < \epsilon)$ and

ii. $0 < D \wedge val(x - a) < D) \vdash 0 < D \wedge val(x - a) < D)$.

2. `Solve-f` is applied to 1(c)i.

3. `SOLVE`$_<$`f` iterates over `AndE, Solve`$_<$`f` on 1(c)ii. For
$V = \boxed{\{(0 < D \wedge val(x - a) < D)\}}$
the resulting new assumptions are

   (a) $V \vdash 0 < D$ which is processed by `Solve`$_<$`f`. This adds $0 < D$ to the constraint store.

   (b) $V \vdash val(x - a) < D$

4. Guided by a control-rule, now `UNWRAPHYP`, `RemoveFocus`, `Mp-b`, `LimHeuristic` are applied in a row.

5. The goal is of the form $\Delta \vdash val(B) < E$. Therefore, the control suggests `UNWRAPHYP` with the focus on the H's subformula $\underline{val(f(x_1) - l_1) < \epsilon_1}$ that is most similar to the goal and that has hypotheses contained in Δ. On the input $H \vdash H$, `IncreaseHyp` outputs the assumption $\Delta \vdash H$.

   (a) `AndE` is applied to the assumption H giving the two assumption

      i. $\Delta \vdash \forall\epsilon_1\exists\delta_1\forall x_1(0 < \epsilon_1 \rightarrow 0 < \delta_1 \wedge val(x_1 - a) < \delta_1 \rightarrow \underline{val(f(x_1) - l_1) < \epsilon_1})$ and

      ii. $\Delta \vdash \forall\epsilon_2\exists\delta_2\forall x_2(0 < \epsilon_2 \rightarrow (0 < \delta_2 \wedge val(x_2 - a) < \delta_2 \rightarrow val(g(x_2) - l_2) < \epsilon_2))$.

   (b) `Skolem-f` is applied to the assumption 5(a)i gives

      i. $\Delta \vdash 0 < E_1 \rightarrow 0 < d_1 \wedge val(X_1 - a) < d_1 \rightarrow \underline{val(f(X_1) - l_1) < E_1}$, and

   (c) `Backchain` on 5(b)i gives

      i. the goal $\Delta \vdash 0 < E_1$

      ii. the assumption $\Delta \vdash 0 < d_1 \wedge val(X_1 - a) < d_1 \rightarrow \underline{val(f(X_1) - l_1) < E_1}$

   (d) `Backchain` on 5(c)ii gives

      i. the goal $\Delta \vdash 0 < d_1 \wedge val(X_1 - a) < d_1$ and

      ii. the assumption $\Delta \vdash \underline{val(f(X_1) - l_1) < E_1}$ with an unwrapped focus. This focus is removed.

6. `MP-b` can now establish the subgoal
$\Delta \vdash val(f(X_1) - l_1) < E_1 \rightarrow val(f(x) + g(x) - (l_1 + l_2)) < \epsilon$.

7. Now `LimHeuristic` is applicable. Its procedure `extract` yields the list and $(1, (g(x) - l_2), [x/x_1])$. `LimHeuristic` yields a new assumption and three subgoals with the new meta-variable M that becomes an argument of the Skolem functions d_1, d_2, x:

   (a) $\Delta \vdash val(1) < M$

   (b) $val(f(X_1) - l_1) < E_1 \vdash val(f(x) - l_1) < div(\epsilon, 2 * M)$

(c) $\Delta \vdash val(g(x) - l_2) < div(\epsilon, 2)$

(d) assumption ($val(f(X_1) - l_1) < E_1 \vdash val(f(X_1) - l_1) < E_1$ (HYP))

8. 7a is satisfied by `Solve`$_<$`b` on M. This adds $val(1) < M$ to the constraint store.

9. 7b is satisfied by SOLVE*: first `Solve*` produces the subgoal $E_1 < div(\epsilon, 2*M) \vee E_1 = div(\epsilon, 2 * M)$ and the subgoal $\emptyset \vdash x = X_1$. The latter is satisfied by `Solve=b` that produces the assumption $\emptyset \vdash x = X_1$. Then `Solve`$_<$`b` closes the first subgoal and adds $E_1 < div(\epsilon, 2 * M)$ to the constraint store.

10. 7c has the following larger subproof guided by control-rules: `UNWRAPHYP`, `RemoveFocus`, and then `SOLVE*`.

    (a) `UNWRAPHYP` iterates over the submethods: `Focus`, `Skolem-f`, `Backchain`. `IncreaseHyp` does not change anything.

        i. `Skolem-f` applied to the assumptions 5(a)ii gives $\Delta \vdash 0 < E_2 \rightarrow 0 < d_2 \wedge val(X_2 - a) < d_2 \rightarrow \underline{val(g(X_2) - l_2) < E_2}$.

        ii. `Backchain` on this assumption yields

            A. the subgoal $\Delta \vdash 0 < E_2$ that is proved later by `Solve`$_<$`b` adding $0 < E_2$ to the constraint store, and

            B. the new assumption $\Delta \vdash 0 < d_2 \wedge val(X_2 - a) < d_2 \rightarrow \underline{val(g(X_2) - l_2) < E_2}$

        iii. `Backchain` decomposes assumption 10(a)iiB into: the subgoal $\Delta \vdash 0 < d_2 \wedge val(X_2 - a) < d_2$ and the assumption $\Delta \vdash \underline{val(g(X_2) - l_2) < E_2}$ which is unwrapped now. The focus is removed.

    (b) This assumption $\Delta \vdash val(g(X_2) - l_2) < E_2$ now serves to prove the subgoal 7c $\Delta \vdash val(g(x) - l_2) < div(\epsilon, 2)$ by SOLVE*: iterates over `Solve*`, `Solve=`, and `Solve*`$_<$`b` which yield the constraint $E_2 < div(\epsilon, 2)$ and the subgoal $\emptyset \vdash x = X_2$ turned into an assumption by `Solve=`.

11. The subgoal from 10(a)iii is decomposed by `AndI` into the subgoals:
    $\Delta \vdash 0 < d_2$
    $\Delta \vdash val(X_2 - a) < d_2$.

    (a) `Solve`$_<$`backward` removes subgoal $0 < d_2$ and adds it to the constraint store because it can be inferred from the existing store.

    (b) $\Delta \vdash val(X_2 - a) < d_2$ satisfied by `SOLVE*`: `Solve*` applied to assumption $V \vdash val(x - a) < D$ and then `Solve*`$_<$`b` (3b) yield the constraint $(D < d_2)$. `Solve*` yields the additional subgoal $\emptyset \vdash x = X_2$ which is changed to an assumption by `Solve`$_=$.

12. The goal 5(d)i $\Delta \vdash 0 < d_1 \wedge val(X_1 - a) < d_1$ is closed by `AndI` (part of SOLVE$_<$) that yields three subgoals:

    - $\Delta \vdash 0 < d_1$ closed by `Solve`$_<$ because it can be inferred from the constraint store.
    - $\Delta \vdash val(X_1 - a) < d_1$ closed by SOLVE*: `Solve*` with the assumption $V \vdash val(x - a) < D$ (3b) and then `Solve*`$_<$`b` passes the goal $D < d_1$ to the constraint store. The additional subgoal $\emptyset \vdash x = X_1$ is removed by `Solve=`.

The final constraint store is:

```
{ [ 0 < E2 < div(E, 2);
    0 < D < D2, D1;
    0 < E1 < div(E, (2 * M)), div(E, 2);
    1 < M < div(E,(2 * E1)) ] }
```

From this state the proof assumption $0 < D < min(d_1 d_2)$ can be extracted for a textbook style proof presentation discussed in the next section.

5 Extracting a Proof from the Plan

Leron [26] points out that there are two ways of presenting a proof that involves constructions: one that *states* the definition of an object and another that provides the *construction* of the object. The latter communicates more information and should, hence, be preferred in teaching proofs. Hence, we want to provide **two different styles** of proof presentation:

1. one that starts with the final constraint state (let ...) and
2. another that captures the constraint solving, i.e., the construction of δ as given by the proof plan (extended by the constraint annotations).

The second (construction style) presentation follows the proof plan whose nodes are annotated by constraint states because this plan is a high-level description on how to find a proof.

The first presentation (textbook style) still needs to be extracted from the annotated proof plan. It is achieved in the following way. The expansion of the proof plan methods yields a proof. This proof has to be augmented by a (conjunctive) assumption line that captures the final state of the constraint store. We mentioned the procedure for extracting this formula Φ in section 3.2. The resulting ND-level proof is the basis for OMEGA's natural language presentation by PROVERB [19] which then produces a textbook style proof with the beginning 'Let δ be ..'. In such a proof presentation, however, the information on how δ has been constructed is not present any more.

6 Experiments and Results

The extended OMEGA can proof plan for limit theorems, including

- LIM+, LIM-,
- LIM*,
- COMPOSITE that states that the composition of two continous functions is continous,
- CONTINOUSifDeriv: a function having a derivative at a point is continuous there,
- SUMcont: the sum of two continous functions is continous,
- PRODUCTcont: the product of two continous functions is continous,

- UNIFcont: a uniformly continous function is continous
- Theorems like $\lim_{x \to a} x^2 = a^2$.

Our hypothesis has been, among others, that the use of control-rules and of supermethods with their particular control makes proof planning feasible. In order to test this hypothesis and to demonstrate the effect of control-rules on the search space, we devised and conducted the following experiments for planning LIM+ and LIM* in the following control contexts:

1. Basic methods only with an intelligent numeric rating; We tested a rating of methods that provides the following order of methods:

   - Same, Same=
   - Solve=
   - Solve$_<$f
   - Solve$_<$b
   - Solve*$_<$b
   - Solve*
   - LimHeuristic
   - AndE, AndI, ImpI, EquivI, Skolem
   - Backchain
   - Mp-b, IncreaseHyp, Focus, RemoveFocus

   OMEGA's proof planner did not succeed in planning a proof for LIM+ using these rated methods but no control-rules. The rating is devised by an experienced human proof planner but still we expect a potentially infinite search because of methods such as `Mp-b`, `Solve`$_<$`f`, and `IncreaseHyp`. This expectation was met. The planner did not succeed with LIM+.

2. Using supermethods; we expect the creation of subplans to reduce the search space in general. We tested the following rating of supermethods and methods that provides the following order of methods:

   - Same, Same=
   - Solve=
   - SOLVE$_<$f
   - SOLVE$_<$b
   - SOLVE*
   - LimHeuristic
   - NORMAL
   - UNWRAPHYP
   - Mp-b, RemoveFocus

OMEGA's proof planner did not succeed in planning a proof for LIM+ using these rated supermethods and methods but no control-rules.

3. Using explicit control-rules; we expect to exclude infinite search by control-rules. This hypothesis proved true as described below.

   The experiments on planning LIM+ and LIM* *with* the control-rules presented in section 3.3 show that the search guided by control-rules succeeded in planning for LIM+ and LIM*. We express the experimental results in terms of search space exploration rather than in terms of CPU time because the current performance of the matcher and the overall planning algorithm is not the topic of this report.

   With the presented control-rules **no** backtracking is necessary to plan LIM+. For planning LIM+, **216 matching attempts** for methods' inputs were performed. For planning LIM*, **336 matching attempts** for methods' inputs were performed. The search performed in supermethods is counted in in this figure. This is a reason for the relatively high number of matching attempts that resulted despite of no backtracking. The matching and the evaluation of the methods' *application condition* restrict again the actual choice of a method. Therefore, no backtracking is necessary in planning LIM+ and LIM*.

The potential availability of all the methods of parent theories is a notable progress. This large set of methods can be restricted by a control-rule that belongs to the *limit* domain.

7 Conclusion and Related Work

Proof planning is an alternative to classical automated theorem proving techniques that has been successfully used in limited areas of theorem proving so far. The vision behind our 'limit' enterprise, which we have chosen as a prototypical area, has been to make proof planning a more widely usable technique in automated theorem proving and to foster even more the application of AI-techniques in theorem proving. Based on AI-planning experience and on the specifity of theorem proving, we have gained a deeper understanding of proof planning and have extended the methodology and the applicability of proof planning to new mathematical domains.

The paper has shown directions to follow in proof planning such as extending the domain knowledge by modularly represented control-rules and by constraint solving, encoding existing heuristics into planning operators, and employing different flexibly applicable planning strategies. Including this domain knowledge seems to be even cognitively adequate since mathematicians specialize in a particular mathematical area and then they have this diverse domain-specific knowledge in addition to general problem solving competence. Integrating domain knowledge into a general problem solving framework is a well established AI principle and more often than not in AI, problem solvers are divided into general reasoners with general heuristics and domain-specific components such as constraint solvers.

The class of limit theorems can be considered a prototype for classes of theorems whose proofs involve the construction of objects or require high-level control.

The fact that we have succeeded in automatically planning a proof of LIM* with our general purpose proof planner shows that these techniques can be successfully used now for

highly nontrivial mathematical theorems. While LIM+ represents the limit of what current theorem provers and planners could handle, LIM* is certainly beyond the capabilities of other current proof planners and theorem provers.

As in most complex planning domains (see, e.g. [38]), the design of methods and control knowledge is crucial for the success of planning and requires a good deal of domain understanding and of knowledge engineering work. We designed methods encapsulating ideas from the special-purpose program [6]. Hence, our proof planning combines a general-purpose proof planner with specific domain knowledge.

Advantages of the proof planning framework are: the resulting high-level, hierarchical representation of proofs, the flexible invocation of methods, the use of explicit global control knowledge, and the possible expansion of plans to checkable proofs. Expanding computations or constructions to a checkable logical proof typically yields proofs of enormous length . That is, in many situations it may be desirable to have a plan only rather than expanding it or constructing calculus-level proofs in the first place.

By using appropriate methods, the theorems and axioms from the theory of $\mathbb{R}$,[12] such as the triangle inequality or the transitivity of $<$, do not need to be referred to explicitly. This seems to be similar to mathematician's way of doing things. This advantage was already mentioned by Bledsoe who qualified proofs without the inclusion of axioms as desirable "because, for most automatic theorem proving programs, the axioms have to be selected by humans for each theorem being proved."[6]

Related Work

Clearly, proof planning in CL^AM is related to proof planning in OMEGA. There are several differences and similarities. For instance, as opposed to our hiding of axioms, CL^AM with colored rippling [39] proved LIM+ by explicitly using the proof assumptions
$(x_1 + x_2) - (y_1 - y_2) = (x_1 - y_1) + (x_2 - y_2)$
$val(x + y) < w \Leftarrow val(x) + val(y) < w$
$x + y < w \Leftarrow x < div(w, 2) \wedge y < div(w, 2)$
$q \rightarrow p_1 \wedge p_2 \Leftarrow (q \rightarrow p_1) \wedge (q \rightarrow p_2)$
$\forall x(p_1 \wedge p_2) \Leftarrow \forall x.p_1 \wedge \forall x.p_2$
$\exists \delta.\Phi(u < \delta \rightarrow p_1 \wedge p_2) \Leftarrow \exists \delta.\Phi(u < \delta \rightarrow p_1) \wedge \exists \delta.\Phi(u < \delta \rightarrow p_2)$
$0 < \epsilon \rightarrow p \wedge q \Leftarrow 0 < div(\epsilon, 2) \rightarrow p \wedge 0 < div(\epsilon, 2) \rightarrow q$,
where some of these assumptions stem from [5].

Similar to the rippling heuristic developed for inductive proofs and used to guide proof planning in CL^AM, some of our control-rules have a focusing purpose. The control-rule mechanism presented is, however, more general than rippling. While rippling [11, 21] is a powerful search heuristic for difference reduction, it is not universal enough for other mathematical proofs. As far as we know, LIM* could not been proved by rippling, for instance.

A new universe appears to opens up when combining theorem proving with constraint solving. This relates particularly to work in constraint logic programming, see [17, 22]. For related work see also [35, 36].

[12] some inherited from parent theories

8 Acknowledgement

Many thanks to the students Alexander Koller, Carsten Ullrich, and Jürgen Zimmer who enthusiastically helped to get OMEGA's proof planning for limit theorems running. In particular, Alexander designed and implemented the constraint solver, Carsten implemented the control-rule interpreter, and Carsten and Jürgen ran the experiments. I thank Jörg Siekmann for suggesting improvements on an earlier draft of this paper.

This work has partially been funded by the Deutsche Forschungsgemeinschaft SFB378.

References

[1] J.R. Anderson. Acquisition of proof skills in geometry. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, pages 191–219. Springer, Berlin, New York, 1984.

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

[3] 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)*, pages 252–255, Townsville, 1997. Springer.

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

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

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

[7] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, San Diego, 1988.

[8] 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.

[9] A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

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

[11] 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.

[12] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *Proc. 10th International Conference on Automated Deduction (CADE)*, volume Lecture Notes in Artificial Intelligence 449. Springer, 1990.

[13] R.B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Computer Science Department, Carnegie Mellon University, January 1995.

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

[15] 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.

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

[17] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, London, 1989.

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

[19] 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.

[20] 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.

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

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

[23] 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.

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

[25] R.E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

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

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

[28] E. Melis. Island planning and refinement. Seki Report SR-96-10, Universität des Saarlandes, FB Informatik, 1996. available from http://www.ags.uni-sb.de/publications/deduktion/seki/SR-96/index.html.

[29] E. Melis and A. Bundy. Planning and proof planning. In S. Biundo, editor, *ECAI-96 Workshop on Cross-Fertilization in Planning*, pages 37–40, Budapest, 1996.

[30] M.Kerber and A.C. Sehn. Proving ground completeness of resolution by proof planning. In *Proceedings of the 10th Florida International AI Conference (FLAIRS-97)*, pages 372–376, 1997.

[31] T. Müller and J. Würtz. *The Constraint Propagator Interface of DFKI Oz.* DFKI, March 1997.

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

[33] V. Saraswat. Cp as a general-purpose constraint-language. In *Proceedings of AAAI-87*, 1987.

[34] A.C. Sehn. DECLAME – eine deklarative Sprache zur Repräsentation von Methoden. Master's thesis, Universität des Saarlandes, 1995. published as SEKI Working Paper SWP-95-02.

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

[36] F. Stolzenburg. Memebership 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.

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

[38] A. Tate, B. Drabble, and R. Kirby. O-plan2: An open architecture for command planning and control. In M.Fox and M. Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[39] 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.

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

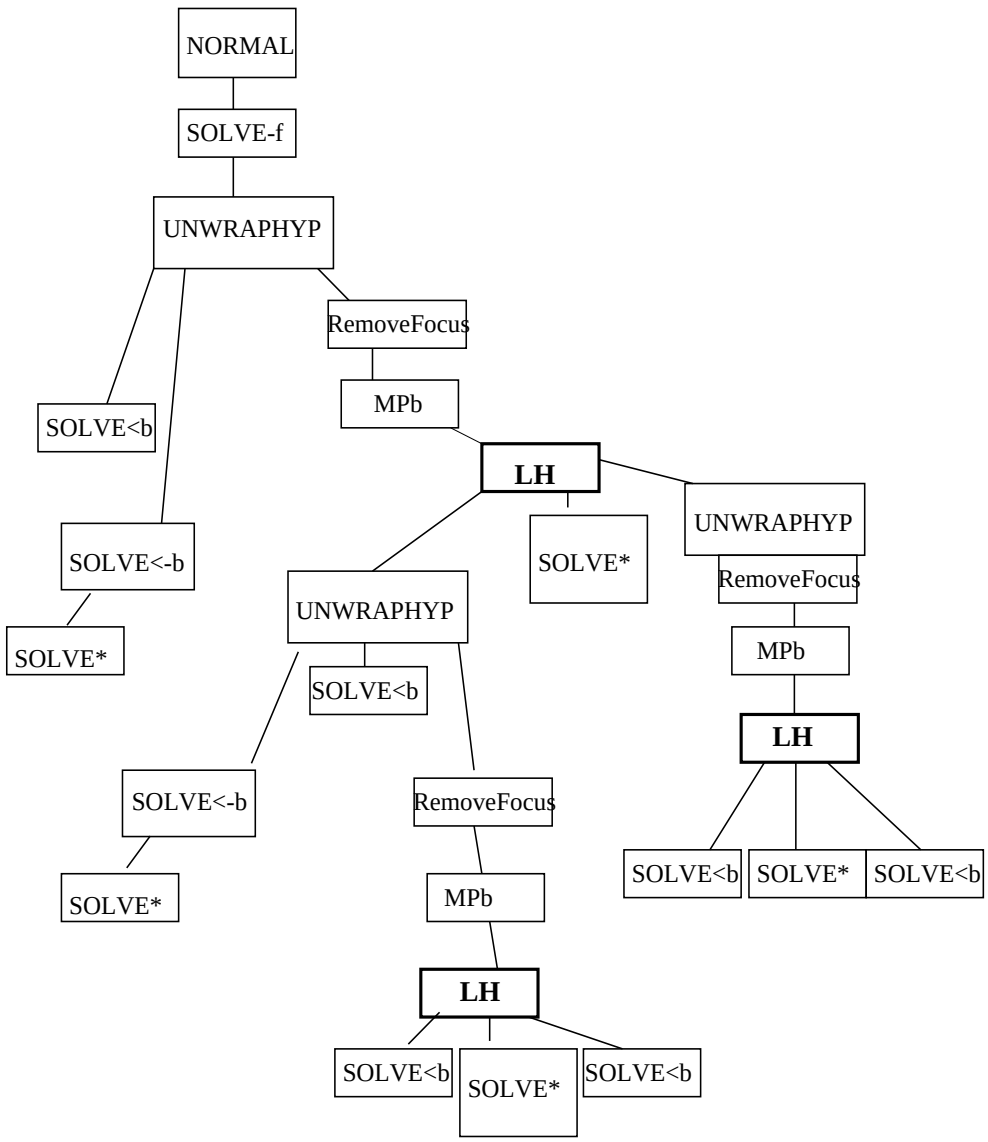

Figure 4: Proof plan of LIM*

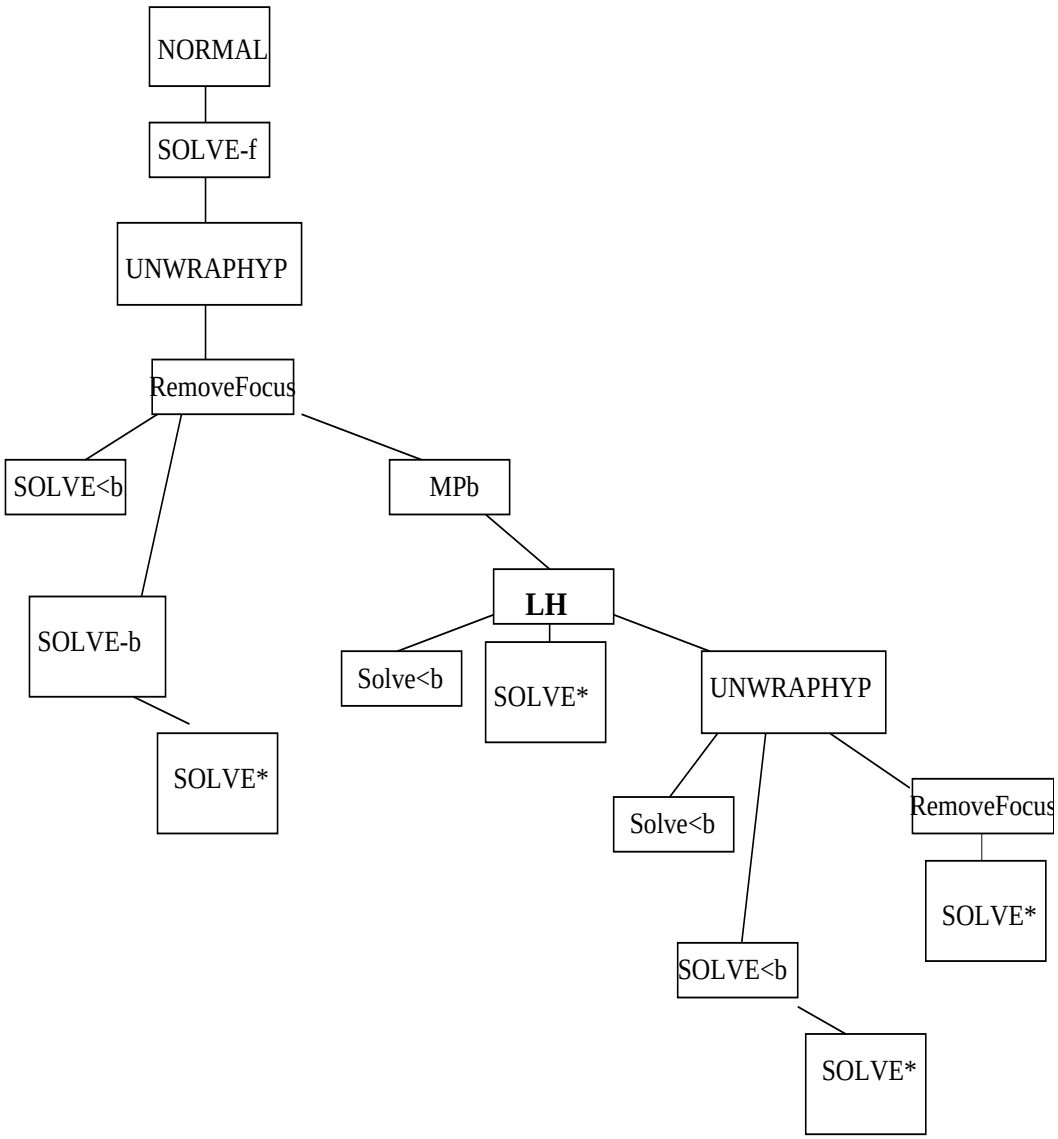

Figure 5: Proof plan of LIM+