Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Artificial Intelligence 172 (2008) 656–684

Artificial Intelligence

www.elsevier.com/locate/artint

Proof planning with multiple strategies

Erica Melis, Andreas Meier, Jörg Siekmann [*]

Universität des Saarlandes and German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany

Received 2 May 2006; received in revised form 13 November 2007; accepted 16 November 2007

Available online 22 November 2007

Abstract

Proof planning is a technique for theorem proving which replaces the ultra-efficient but blind search of classical theorem proving systems by an informed knowledge-based planning process that employs mathematical knowledge at a human-oriented level of abstraction. Standard proof planning uses *methods* as operators and *control rules* to find an abstract proof plan which can be expanded (using *tactics*) down to the level of the underlying logic calculus.

In this paper, we propose more flexible refinements and a modification of the proof planner with an additional strategic level of control above the previous proof planning control. This *strategic* control guides the cooperation of the problem solving strategies by meta-reasoning.

We present a general framework for proof planning with multiple strategies and describe its implementation in the MULTI system. The benefits are illustrated by several large case studies, which significantly push the limits of what can be achieved by a machine today.

Keywords: Theorem proving; Proof planning; Blackboard architecture; Planning; Meta-reasoning

1. Introduction

The control problem, i.e. how to choose one of the many potential actions an intelligent agent—man or machine alike—has at its disposal, is fundamental to all problem solving processes. It stimulated the development of many software architectures in artificial intelligence, including blackboard architectures [18,19,24] and multi agent systems [54].

In spite of their increasing sophistication, however, many systems are still rather inflexible and employ a predetermined and fixed control schema. In particular, this is true for most classical automated theorem proving systems which expand and efficiently search through very large search spaces guided by pre-fixed general-purpose filters and heuristics. A modification on the fly or a flexible combination of different heuristics to tackle sub-problems is in general not possible. As a result, these systems can not recognize mathematically promising search paths as they go, and they make up for this deficiency by their sophisticated representational techniques (see Chapter VIII in [45]) and

[*] Corresponding author.
E-mail addresses: melis@dfki.de (E. Melis), meier@dacos.com (A. Meier), siekmann@dfki.de (J. Siekmann).
URLs: http://www.ags.uni-sb.de/~melis (E. Melis), http://www-ags.dfki.uni-sb.de/JS/index.html (J. Siekmann).

general search heuristics to blindly examine a space of up to several billion nodes. The final performance of a system, however, depends on whether the pre-selected search heuristic is appropriate for the particular problem at hand.[1]

As a reaction to these problems several remedies have been tried:

(1) Different systems are competitively combined such that machine resources are allocated to the most promising system at a time [2,47,51,59]. Typical meta-heuristics for system selection and resource allocation evaluate characteristics of the problem at hand and compare it to past experience. For instance: "in the past, the system S with heuristics H was best suited for problems with the following number and type of clauses and/or equations".
(2) Cooperation of different systems, which exchange intermediate results [15], where meta-heuristics are used to decide which results to exchange. For example: "The derived clause C is a unit clause, which could be useful in another system and hence, should be exported".

Proof planning [7] is a technique for theorem proving in which proofs are planned at a higher level of abstraction, where individual choices can be mathematically motivated by the semantics of the domain. Thus, proof planning swings the pendulum from the desert of the blind but ultra-efficient search-based paradigm of classical automated theorem proving to the green grass of knowledge based systems. In particular, proof planning tackles theorems not only using general logic based methods but also by using domain-specific and general mathematical knowledge, encoded explicitly into methods and control [39,41]. Essentially, however, proof planners like CLAM or ΩMEGA, are monolithic, in the sense that the planning algorithm is pre-defined and cannot take full advantage of the runtime knowledge that is available during the problem solving process.

Our experiments with proof planning in the past decade indicate that the search process would benefit substantially from more flexibility of choice and more usage of runtime knowledge instead of a mere competitive application of several systems or the simple exchange of intermediate results. This situation has been recognized in other real-world applications of planning as well, see, e.g., [56].

In the following, we report our results for *proof planning with multiple strategies*, which is built upon three general principles: (1) *decomposition* of the monolithic search process into independent processes; (2) *structuring* the set of methods and the corresponding control knowledge into strategies; (3) *meta-reasoning* with explicitly represented control knowledge at a *strategic level*.

After more than a decade of development and experimentation this article presents our current 'final' stable solution, introducing conceptually clean notions and notation and finally backs it all up with several large case studies. Most case studies had been published before at conferences and workshops on automated theorem proving, but stepping back now from the presentation at the time of those particular achievements we like to present and summarize our current more general view in this journal paper.

The remainder of this paper is organized as follows. After a brief introduction to proof planning and some motivating examples, we show how the three principles above are implemented in the MULTI system [32,35]. The results are then discussed and evaluated with several case studies, which illustrate the potential of explicitly represented strategic knowledge and control and demonstrate what can be achieved with this automated reasoning system today.

2. Preliminaries

ΩMEGA is a theorem proving environment developed at the University of Saarbrücken based on proof planning and other techniques and MULTI is now the proof planner of the current system. The ΩMEGA project [48] represents one of the major attempts to build an all encompassing assistant tool for the working mathematician or the software engineer, which combines interactive and automated proof construction for domains with rich and well-structured mathematical knowledge. The inference mechanism at the lowest level of abstraction is based on a higher order natural deduction (ND) variant of a soft-sorted version of Church's simply typed λ-calculus [10]. While this represents the "machine code" of the system the user will seldom want to see, the search for a proof is conducted at a higher level of abstraction by the proof planning process.

[1] The general wisdom is: no single theorem prover or heuristic is best for all problems, see contest http://www.cs.miami.edu/~tptp/CASC/.

Proof planning differs from traditional search-based techniques in automated theorem proving as the proof of a theorem is planned at an abstract level, where an outline of the proof is constructed first. This outline, that is the abstract proof plan, can be recursively expanded with methods and tactics eventually down to a logical calculus proof. Most plan operators, called *methods* for this reason, represent mathematical techniques familiar to a working mathematician.

Knowledge-based proof planning [41] employs even more techniques from artificial intelligence such as hierarchical planning, constraint solving and control rules, which encode the "how to solve it" knowledge for guiding the search at an abstract level. While the knowledge of a mathematical domain represented by operators and control rules can be specific to the mathematical field at hand, the representational techniques and reasoning procedures are general-purpose.

The plan operators in mathematical proof planning are called *methods*. They (partially) describe changes of proof states by pre- and postconditions which are called *premises* and *conclusions* in the following. The premises and conclusions of a method are formulae (more precisely: sequents) in a higher-order language and the conclusions are considered as logically inferable from the premises.

Hence, a mathematical theorem proving problem is expressed as a planning problem whose initial state consists of the proof assumptions and the goal description which is the conjecture to be shown. Proof planning searches for a sequence (or a hierarchy) of instantiated methods, a *solution plan*, which transforms the initial state with assumptions into a state containing the conjecture.

A proof planner can be realized by the following search procedure:

- As long as there are goals, select a goal and try to apply a method to it.
- If there is a goal for which no method is applicable, then backtrack to the method application, which introduced this goal.
- When all goals are closed, employ a constraint solver to instantiate the variables within the methods.

The reason for the late variable instantiation is to first collect as many individual constraints as possible (as long as there are goals) and instantiate all variables to satisfy the collected constraints only at the end.

As opposed to precondition achievement planning [55], effects of methods in proof planning usually do not cancel each other in proof planning. For instance, a method applied to an open node with effect $\neg F$ does not threaten the effect F introduced by another method for another open node. Dependencies among open nodes may result from shared variables for terms and from their constraints. Constraints range over the mathematical domain and have the form $x < c$, $y + b < d$, etc. The constraints created during the proof planning process are collected in a constraint store.

Methods, control rules, and strategies

In order to make the ingredients of proof planning more explicit, let us repeat what methods and control rules contribute to proof planning and then extend the discussion towards our new notions.

Methods have been perceived by Alan Bundy as tactics augmented with preconditions and effects. In our terminology, a method represents an inference of the conclusion from the premises. Backward methods reduce a goal (the conclusion) to new goals (the premises). Forward methods, in contrast, derive new conclusions from given premises.

Control rules represent mathematical knowledge on how to proceed in a particular mathematical situation, and they guide the proof planning process. They influence the planner's behavior at choice points (e.g., which goal to tackle next or which method to apply next) by preferring certain members from the list of possible goals or from the list of possible methods. This way, promising search paths are preferred and the general search space can be pruned substantially.

Methods and control rules are the main ingredients of current proof planning systems, but they do not always provide enough structure and flexibility for the problem solving process as the past decade of our experimentation revealed. First, there is a problem with the planning algorithm itself, which cannot be decomposed into its main functionalities nor can new techniques easily be added. This will be discussed in more detail in Section 4. Secondly, structuring the knowledge can reduce the huge search space. For instance, if we want to prove a continuity theorem in the theory of analysis, it makes a difference whether we prove it via limit theorems, via an epsilon-delta technique or via converging sequences. Similarly, human experts—here a mathematician—has a variety of different strategies at her disposal to tackle such specific problems. This structuring functionality is addressed by certain strategies we define below and moreover, by a context. Some strategies apply in a specific mathematical field only, while others are more general purpose.

A *strategy* as it is now used in our multi-strategy proof planning employs a specific refinement or modification algorithm and a subset of methods and control rules. This subset has to be typical for the particular mathematical proof technique we want to simulate. For instance, one strategy may use an external computer algebra system for some computation, another one may attack the problem with a completely different set of methods such as the epsilon-delta techniques or the methods and control rules may pertain for a proof by induction. Furthermore, the strategy may cooperate with another strategy from a different theory or it may use a completely different backtracking technique.

Meta-reasoning about which strategy to employ for a given problem introduces an additional explicit choice point different from and above the current control and, thus, the system searches also at the level of strategies, which is the subject of this paper.

Finally, proof planning takes place in the particular *context* of a mathematically theory, as discussed in Section 8.

3. Motivating examples

Our first example shows a proof for the theorem stating that the sum of the limits of two functions equals the limit of their sums (referred to as `Lim+` in what follows), which had been solved by other (earlier) proof planning systems already. It is presented here for motivation in order to understand problems and proof plans in the domain of epsilon-delta proofs.This introductory example is followed by two examples which cannot be solved by previous systems. They demonstrate the need for a decomposition of the proof planning process and for using runtime information.

3.1. The `Lim+` *problem*

The theorem claims that the limit (at the point $x = a$) of the sum of two functions f and g equals the sum of their limits. Hence, we have two assumptions, which define the limit of f and g. This definition has the usual form of a limit, saying that for arbitrarily small values of ϵ there exists a δ-environment (of the x-argument) for which all function-values are within the ϵ-environment of limit l. Hence these proofs are often called ϵ–δ-proofs, as they have a typical structure familiar to most first year students of maths.

$$\lim_{x\to a} f(x) \equiv \forall\epsilon_1\big(0<\epsilon_1 \Rightarrow \exists\delta_1\big(0<\delta_1 \wedge \forall x_1\big(0<|x_1-a|<\delta_1 \Rightarrow \big|f(x_1)-l_1\big|<\epsilon_1\big)\big)\big) \tag{1}$$

and

$$\lim_{x\to a} g(x) \equiv \forall\epsilon_2\big(0<\epsilon_2 \Rightarrow \exists\delta_2\big(0<\delta_2 \wedge \forall x_2\big(0<|x_2-a|<\delta_2 \Rightarrow \big|g(x_2)-l_2\big|<\epsilon_2\big)\big)\big). \tag{2}$$

The conjecture of `Lim+` is:

$$\forall\epsilon\big(0<\epsilon \Rightarrow \exists\delta\big(0<\delta \wedge \forall x\big(0<|x-a|<\delta\big)\big)\big). \tag{3}$$

A typical proof for this and similar conjunctures requires appropriate expressions for δ. How to invent these is what a maths student learns usually in his first semester on analysis. Here is one of the standard proofs taken from R. Bartle, D. Sherbert: "Introduction to Real Analysis" (1982) from which we have taken most of the ϵ–δ theorems to be proved. This standard proof does not presuppose extra lemmata, except the Triangle Inequality

$$|a+b| \leqslant |a|+|b|.$$

The `Lim+`-theorem is Theorem 4.2.4 in [4].

Theorem 4.2.4. *Let* $\lim_{x\to a} f(x) = l_1$ *and* $\lim_{x\to a} g(x) = l_2$ *then* $\lim_{x\to a}(f(x)+g(x)) = l_1 + l_2$.

Proof. In order to show the theorem, we need to estimate the magnitude of

$$\lim_{x\to a} f(x) + \lim_{x\to a} g(x) - (l_1 + l_2).$$

Using the Triangle Inequality we obtain

$$\Bigl|\Bigl(\lim_{x\to a} f(x) + \lim_{x\to a} g(x)\Bigr) - (l_1 + l_2)\Bigr| \tag{4}$$

$$= \Bigl|\Bigl(\lim_{x\to a} f(x) - l_1\Bigr) + \Bigl(\lim_{x\to a} g(x)\Bigr) - l_2\Bigr| \tag{5}$$

$$\leqslant \Bigl|\lim_{x\to a} f(x) - l_1\Bigr| + \Bigl|\lim_{x\to a} g(x) - l_2\Bigr|. \tag{6}$$

Now by the definitions of lim for the functions f and g, for all $\frac{\epsilon}{2}$ there exists δ_1 and for all $\frac{\epsilon}{2}$ there exists δ_2 such that for $|x-a| < \delta_1$ and $|x-a| < \delta_2$, i.e.

$$\bigl|\lim f(x) - l_1\bigr| < \frac{\epsilon}{2},$$

$$\bigl|\lim g(x) - l_2\bigr| < \frac{\epsilon}{2}$$

and hence

$$\bigl|\lim f(x) - l_1\bigr| + \bigl|\lim g(x) - l_2\bigr| < \epsilon.$$

Thus for $\delta = \min(\delta_1, \delta_2)$ if $|x-a| < \delta$, then $|(\lim f(x) + g(x)) - (l_1 + l_2)| < \epsilon$ and hence by definition

$$\lim_{x\to a}\bigl(f(x) + g(x)\bigr) = (l_1 + l_2) = \lim_{x\to a} f(x) + \lim_{x\to a} g(x). \qquad \square$$

It is the proof the MULTI system has found too, because COMPLEXESTIMATE essentially captures this kind of recipe. Actually, the "recipe" is even more general and includes more than an application of the Triangle Inequality. We shall discuss the machine generated proof further on below

This general recipe, namely how to decompose (in)equality goals in order to end up with simple inequalities that determine restrictions for δ depending on ϵ is encapsulated in our method COMPLEXESTIMATE. It is shown in its frame representation below as usual with slots and slot fillers.

method: COMPLEXESTIMATE(a, b, e_1, ϵ)			
premises	(0), ⊕(1), ⊕(2), ⊕(3), ⊕(4)		
conclusions	⊖ L12		
application condition	there is a unifier σ of the terms a and b, and $\exists k, l(casextract(a_\sigma, b) = (k, l))$ and $b = k * a_\sigma + l$		
proof schema	(0).	$\Delta \vdash \lvert a\rvert < e_1$	()
	(1).	$\vdash \lvert a_\sigma\rvert < \epsilon/(2 * \mathbf{V})$	(OPEN)
	(2).	$\Delta \vdash \lvert k\rvert \leqslant \mathbf{V}$	(OPEN)
	(3).	$\vdash 0 < \mathbf{V}$	(OPEN)
	(4).	$\Delta \vdash \lvert l\rvert < \epsilon/2$	(OPEN)
	L0.	$\vdash b = b$	(Axiom)
	L1.	$\vdash b = k * a_\sigma + l$	(CAS;L0)
	.	$\vdash \ldots$	(...)
	L12.	$\Delta \vdash \lvert b\rvert < \epsilon$	(inferred;L1,(0),(1),(2),(3),(4))

Let us explain the method's representation in order to provide a flavor for methods more generally: the method has the *name* COMPLEXESTIMATE with parameters a, b, e_1 and ϵ. The slot *premises* has as slot fillers the line (0), i.e. the formula (in a sequence calculus) $\Delta \vdash |a| < e_1$, and lines (1), (2), (3) and (4) of the *proof schema*. The ⊕ indicates these lines are to be added to the proof state. The *conclusion* has line L12, i.e $\Delta \vdash |b| < \epsilon$, as its slot filler and the ⊖ indicates that a line is to be removed from the state. There is also a slot for a (meta-level *application condition* and, finally, the operationalization of this method as a partial proof is captured in the *proof schema*. The *application condition* of COMPLEXESTIMATE requires that a and b can be unified with the substitution σ and that a decomposition $b = k * a_\sigma + l$ can be found (with the help of a computer algebra system). The terms k and l resulting from that computer algebra computation are used in the subgoals (2) and (4) produced by the application of COMPLEXESTIMATE. The *proof schema* contains proof lines. The lines with the justification OPEN are the new subgoals. In COMPLEXESTIMATE these are the lines (1), (2), (3) and (4). The goal that is closed by this method (i.e. L12) can be inferred from these subgoals, using some further axioms and theorems, which are indicated in the justification. Other lines may have justifications pointing to a computation (e.g., L1 refers to a computation by a computer algebra system (CAS) or axioms and theorems (e.g., line L0). When the proof plan is expanded, the *proof schema* is inserted into the final proof.

This recipe essentially replaces the goal $\Delta \vdash |b| < \epsilon$ (line L12 in the frame) by "simpler" subgoals (lines (1), (2), (3) and (4)) in case there is an assumption $\Delta \vdash |a| < e_1$ (line (0)) in the state. These new goals are "simpler" inequalities because they contain only some constants and the terms k and l which result from the decomposition of the original complex term b and the variable $\mathbf{V}$ later to be replaced by a term.

Now, when proof planning is applied to the limit conjecture and the two assumptions above, the proof planner has in general several methods at its disposal which could be applied. In our case there are methods which first decompose the conjecture and the assumptions. Among others, this yields the new assumptions[2] $|f(v_{x_1}) - l_1| < v_{\epsilon_1}$ and $|g(v_{x_2}) - l_2| < v_{\epsilon_2}$ and the two new goals $0 < v_\delta$ and $|(f(c_x) + g(c_x)) - (l_1 + l_2)| < c_\epsilon$.[3] The first goal, $0 < v_\delta$, is taken care of by the method TELLCS which closes the goal and adds it to the constraint store of the constraint solver COSIE [62]. The second goal $|(f(c_x) + g(c_x)) - (l_1 + l_2)| < c_\epsilon$ requires further decomposition, which is now done by the method COMPLEXESTIMATE.

The actual application of COMPLEXESTIMATE to this goal matches the method's parameters a, b, e_1 and ϵ as follows:

$$\epsilon \mapsto c_\epsilon, \quad b \mapsto \big(f(c_x) + g(c_x)\big) - (l_1 + l_2), \quad a \mapsto f(v_{x_1}) - l_1, \quad \text{and} \quad s_1 \mapsto v_{\epsilon_1},$$

i.e., the goal $|(f(x) + g(x)) - (l_1 + l_2)| < c_\epsilon$ is instantiated for $|b| < \epsilon$ and the assumption $|f(v_{x_1} - l_1)| < v_{\epsilon_1}$ for $|a| < e_1$.

This concrete application of COMPLEXESTIMATE computes the values $k = 1$ and $l = g(c_x) - l_2$; it deletes the goal $|(f(c_x) + g(c_x)) - (l_1 + l_2)| < c_\epsilon$ and yields four new goals:

$$\epsilon_1 < \frac{c_\epsilon}{2 * \mathbf{V}}, \tag{7}$$

$$|1| \leqslant \mathbf{V}, \tag{8}$$

$$0 < \mathbf{V}, \tag{9}$$

$$\big|g(c_x) - l_2\big| < \frac{c_\epsilon}{2}. \tag{10}$$

Lines (7), (8), (9) can be closed by TELLCS because they are already in the form of a constraint. Goal (10) is reduced by a method called SOLVE*, which uses the derived assumption $|g(v_{x_2}) - l_2| < v_{\epsilon_2}$ from above. The method SOLVE* exploits the transitivity of the relation $<$ (as well as of $>, \leqslant, \geqslant$) and reduces a goal of the form $a_1 < b_1$ (or $a_1 > b_1$) to a new goal $\sigma b_2 \leqslant \sigma b_1$ ($\sigma b_2 \geqslant \sigma b_1$) in case there is an assumption of the form $a_2 < b_2$ ($a_2 > b_2$) in the proof state and a_1, a_2 can be unified with the substitution σ.

The resulting goals $v_{\epsilon_2} \leqslant \frac{c_\epsilon}{2}$ and $v_{x_2} = c_x$ can be closed by TELLCS. The decomposition of the assumptions for f and g lead to some further goals, which are all solved in the subsequent proof planning process.

When all goals are closed, the constraint solver COSIE computes appropriate instances for variables that are consistent with the collected constraints so far. In this case, it generates the following instantiation:

$$v_\delta \mapsto \min(c_{\delta_1}, c_{\delta_2}), \quad v_{\epsilon_1} \mapsto \frac{c_\epsilon}{2}, \quad v_{\epsilon_2} \mapsto \frac{c_\epsilon}{2}.$$

An interesting point to note is, that these happen to be the same values as used in the human proof of `Lim+` in the textbook of Bartle and Sherbert [4] and a proof presentation technique based on proof plans [40] would present just that proof, even showing the collection of restrictions for δ as they enter the constraint store.

Now, we contrast the `Lim+` theorem and the search for its proof with two other ϵ–δ-proofs for which previous proof planning systems fundamentally fail. Both examples are taken from the same textbook on analysis.

[2] Notation: proof planning replaces quantified variables either by constants or placeholder variables. The placeholder variable that substitutes a quantified variable x is denoted by v_x. The constant substituted for a quantified variable x is denoted by c_x.

[3] During the decomposition of the assumptions further goals are created and the decomposition of the conjecture yields further assumptions. In order to illustrate the basic proof planning approach we ignore these details.

3.2. *The* `ContIfDerivative` *problem*

The `ContIfDerivative` theorem states that, if a function f has a derivative f' at point a, then f is continuous at a (Theorem 6.1.2 in [4]). The proof planning problem consists of the assumption, which says that f has a derivative f' at point a:

$$\forall\epsilon_1\left(0<\epsilon_1\Rightarrow\exists\delta_1\left(0<\delta_1\wedge\forall x_1\left(0<|x_1-a|<\delta_1\Rightarrow\left|\frac{f(x_1)-f(a)}{x_1-a}-f'\right|<\epsilon_1\right)\right)\right)$$

and the actual conjecture, which is the definition for continuity of f:

$$\forall\epsilon\big(0<\epsilon\Rightarrow\exists\delta\big(0<\delta\wedge\forall x\big(|x-a|<\delta\Rightarrow\big|f(x)-f(a)\big|<\epsilon\big)\big)\big).$$

Standard proof planning for `ContIfDerivative` fails, because a goal resulting from some side condition cannot be proved—although it is true in this context—and backtracking alone does not find a solution plan either. This is not just a technical weakness but a principle failure of a system which cannot use runtime knowledge.

We shall now drive more directly to the weak spot, without all the details necessary to understand the whole proof planning process. The decomposition of the conjecture and assumption works just as before in `Lim+` and results now in the goal

$$\big|f(c_x)-f(a)\big|<c_\epsilon \tag{11}$$

and a new assumption

$$\left|\frac{f(v_{x_1})-f(a)}{v_{x_1}-a}-f'\right|<v_{\epsilon_1}. \tag{12}$$

Using this assumption the goal (11) can be shown in several steps. That is, the method COMPLEXESTIMATE is applied to the goal (11) as in the proof of `Lim+` and all resulting goals can be closed. Unfortunately, a goal $|c_x-a|>0$ is also generated as a side condition during the decomposition of the initial assumption. This subgoal cannot be proved since it is not true in general.

An analysis of this situation reveals however that we could introduce a case split with the cases $|c_x-a|>0$ $|c_x-a|\leqslant 0$, and $|f(c_x)-f(a)|<c_\epsilon$ can now be proven under both conditions $|c_x-a|>0$ as well as $|c_x-a|\leqslant 0$.

But where should this case split be introduced? An *a priori* introduction of the case split is not feasible since neither the need for it nor the cases themselves are known. Only the impasse provides the information for its need and only the subsequent analysis shows how to modify the overall proof plan in order to circumvent this problem. Such a knowledge-based analysis, i.e., runtime information of the impasse and a flexible modification of the proof plan is not possible in standard proof planning. This problem has also been noticed by Alan Bundy and his students and they developed another approach to this kind of impasse driven analysis (we discuss their technique based on *critics* in Section 7.2).

3.3. *The* `LimPlusConst` *problem*

The theorem `LimPlusConst` states that the limit of a function $f(x+c)$ at $x=0$ equals l, if the limit of the function $f(x)$ at $x=c$ equals l (Exercise 4.1.3 in [4]).

The problem consists of the assumption $\lim_{x\to c}f(x)=l$, i.e.,

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

and of the conjecture $\lim_{x\to 0}f(x+c)=l$, i.e.,

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

The proof planner first decomposes the conjecture and the assumption as in the proof above. This yields the new assumption $|f(v_{x_1})-l|<v_{\epsilon_1}$ and two new goals $0<v_\delta$ and $|(f(c_x+c)-l)|<c_\epsilon$ from the conjecture. The first goal is directly closed by TELLCS since it is just a constraint on v_σ. The second goal is tackled with the method SOLVE*, which uses the new assumption $|f(v_{x_1})-l|<v_{\epsilon_1}$.

Hence, this results in two new goals:

$$v_{\epsilon_1} \leqslant c_\epsilon, \tag{13}$$

$$v_{x_1} = c_x + c. \tag{14}$$

Now TELLCS is applicable and sends these two (in)equalities to the constraint solver. Continuing the planning process, the decomposition of the initial assumption also yields the goals:

$$|v_{x_1} - c| > 0, \tag{15}$$

$$|v_{x_1} - c| < c_{\delta_1} \tag{16}$$

and the decomposition of the initial conjecture yields the assumptions $|c_x| > 0$ and $|c_x| < v_\delta$. Now, the two goals (15) and (16) follow mathematically from these two assumptions but unfortunately SOLVE* cannot be applied to these goals and inequality assumptions since the terms $v_{x_1} - c$ and c_x cannot be unified.

A more intelligent instantiation, i.e., an eager[4] instantiation of the variable v_{x_1} by $c_x + c$ could unblock the planning process since the goals (15) and (16) would be instantiated to $|c_x + c - c| > 0$ and $|c_x + c - c| < c_{\delta_1}$, which can be simplified to $|c_x| > 0$ and $|c_x| < c_{\delta_1}$ and hence, can be closed by SOLVE* using the assumptions $|c_x| > 0$ and $|c_x| < v_\delta$.

Thus, we need a variable instantiation *on-demand* rather than the usual schedule, where this is done at the end of the proof planning process.

In summary, the hard-wired planning algorithm with *method application*, *backtracking* and *variable instantiation* sometimes impedes mathematically motivated proof constructions and their flexible combination. For this reason, we propose first to decompose the proof planner such that these three operations become independent. Secondly we shall now introduce the new notion of a *strategy* as mentioned in Section 2 (preliminaries). Apart from other considerations, which led to this new and more abstract level of control, it is also used to control the application of the three components, i.e. method application, backtracking and variable instantiation.

4. Proof planning with multiple strategies

Our monolithic proof planner in ΩMEGA was decomposed into a problem solving process where a flexible combination and cooperation of various problem solving operations are guided by explicit strategic control knowledge for the respective mathematical domains.

This is achieved by:

- a *decomposition* of the planner into independent algorithms
- making *strategies* first class citizens, which use these algorithms as well as specific methods and their control rules
- adding new algorithms (e.g., the call of external systems) and their related strategies
- *meta-reasoning* about their application at a strategic level of control.

Decomposition

The decomposition of the functionalities of the original planner gives us separate and independent algorithms for different plan refinements and modifications that can then be combined in various ways. These new algorithms are also generalized and extended beyond their original functionality. Moreover, additional algorithms can be introduced as well, even if they did not belong to the original core functionalities of the proof planner.

More concretely, we have decomposed the original planner into the following procedures:

PPlanner which refines a proof plan by applying methods
InstVar which refines a proof plan by instantiating variables
BackTrack which modifies a plan by backtracking to a previous choice point.

The following three algorithms are new and have also been added to the framework:

4 Before all goals are closed.

Exp	which refines a proof plan by expanding complex steps
ATP	which refines a proof plan by calling an external theorem proving system to solve a subproblem, as described in more detail in one of the case studies in Section 6.2
CPlanner	which refines a proof plan by transferring steps from a source proof plan to a target proof plan.

To simplify matters, we shall focus on **PPlanner**, **InstVar** and **BackTrack** in the remainder of this article since they are most relevant for the case studies in Section 6.

Strategies

A strategy is defined by an algorithm and its parameter instantiations. It determines the particular search behavior while it is active. Each strategy is of a specific type; for the present description it is either of type **BackTrack**, **InstVar**, **ATP**, or **PPlanner**.

The procedure for backtracking can be instantiated with different techniques for backtracking. Different strategies of type **BackTrack** realize different kinds of backtracking such as chronological backtracking or different kinds of intelligent goal-directed backtracking. Different **InstVar** strategies can either employ constraint solvers, computer algebra systems or other external reasoning systems to compute a suitable instantiation for a variable.

The generalized **InstVar** algorithm employs several means to instantiate variables which is controlled by the particular **InstVar** strategies.

PPlanner strategies are used to insert *methods* into a proof plan. For instance, the examples mentioned in Section 6.1 are typical for ϵ–δ-proofs, which involve reasoning with inequalities. Hence, one of the **PPlanner** strategies is called SolveInequality; it is defined below in Section 5.1 for illustration.

Meta-reasoning about control

A set of different strategies, is only one ingredient of a more sophisticated overall problem solving behavior. Another ingredient, just as important, is the flexible control that guides the application and combination of these strategies. The flow of control is not (always) pre-defined but computed as we go. This is a difficult task and we shall use a blackboard architecture to implement meta-reasoning about the choice of the 'right' strategy.

For (human) mathematical problem solving, Schoenfeld [46] makes a similar observation:

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

5. Implementation

We shall now present the technical realization of proof planning with multiple strategies in the MULTI planner of the new ΩMEGA system. First, we discuss strategies and strategic control reasoning and then explain the architecture of the implementation.

5.1. Strategies

As mentioned before a strategy encapsulates a certain problem solving behavior, i.e. in our context a particular way to prove a theorem. Technically, a strategy is a condition-action pair. The condition describes the legal conditions for its applicability and the action includes the algorithms that the strategy employs as well as the values for its parameters. Strategies are represented as frame like data-structures and Figs. 1–4 show some examples.

The overall theorem proving job is driven by a dynamic listing of *tasks*. Strategies respond to tasks which include open goals to be closed, variable instantiations to be computed, lemmas to be proven and other demands. This will be elaborated in the discussion of the architecture in Section 5.3.1.

Strategy: SolveInequality		
Condition	*inequality-task*	
Action	Algorithm	PPlanner
	Methods	COMPLEXESTIMATE, TELLCS, SOLVE*, ...
	Control-Rules	`prove-inequality`, `eager-instantiate`, ...
	Termination	*no-inequality-tasks*

Fig. 1. The SolveInequality strategy.

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

Fig. 2. The **InstVar** strategy InstIfDetermined.

PPlanner *strategies*

A **PPlanner** strategy consists of a set of methods, a list of control rules and a termination condition. When a **PPlanner** strategy is invoked, the **PPlanner** algorithm works only with those methods that are specified in the strategy. Similarly it evaluates only the specific control rules. Hence, **PPlanner** strategies provide a means to structure the set of all methods and their control. A strategy terminates its operation, when the termination condition is satisfied.

The control within a strategy is determined by its control rules at the usual choice points where a goal or a method is selected. However, the **PPlanner** algorithm sets also a choice point, which can be modified by the control rules, in order to interrupt its processing, even if the termination condition is not yet satisfied. This is an important feature we are using for the cooperation of strategies: a strategy can be suspended at this "inner choice point" to call another strategy.

As an example of a **PPlanner** strategy consider SolveInequality in Fig. 1. SolveInequality employs the methods and control rules used for proof planning ϵ–δ-proofs. That is, it captures the common proof patterns and heuristics for ϵ–δ-proofs and for other problems that involve the manipulation/reduction of (in)equalities over real numbers. This application situation is expressed by its condition *inequality-task*. Its termination condition holds as soon as there are no equalities or inequalities goals left.

InstVar *strategies*

The application of a method (in a **PPlanner** strategy) leads to (partial) instantiation of variables as the result of matching the parameters of the method with the current open goal and assumptions.

Now, the **InstVar** strategies provide additional means to instantiate variables using a constraint solver, a computer algebra system or any other "oracle" to compute a proper instantiation. The algorithm **InstVar** has one parameter, which is a function that computes the instantiation for a variable.

For instance, variables that occur in constraints collected by the constraint solver CoSIE can be instantiated either by InstIfDetermined or ComputeInstFromCS (see Figs. 2 and 3 respectively). The instantiation differs with respect to the current state of the constraint solver. InstIfDetermined is applicable, if the instantiation of the variable is already uniquely determined by the constraints collected so far and this unique instantiation is then carried out. ComputeInstFromCS is applicable otherwise to any instantiation task for variables for which constraints are collected. Its computation function requests the constraint solver to return some instantiation that is consistent with all constraints collected so far.

Strategy: ComputeInstFromCS		
Condition	*var-in-cs*	
Action	Algorithm	InstVar
	Function	*compute-consistent-instantiation*

Fig. 3. The **InstVar** strategy ComputeInstFromCS.

Strategy: BackTrackStepToTask		
Condition	*not-root-goal*	
Action	Algorithm	BackTrack
	Function	*stepback-to-task*

Fig. 4. The BackTrackStepToTask strategy.

```
(control-rule delay-ComputeInstCosie
              (kind strategic)
              (IF (and (goal-tasks)
                       (job-offer ComputeInstFromCS JO)))
              (THEN (reject JO)))
```

Fig. 5. The strategic control rule `delay-ComputeInstCosie`.

Other possible function parameters of **InstVar** call a computer algebra system (or any other mathematical calculation package) to compute a value for the variable.

BackTrack *strategies*

The parameter of this strategy is a function which computes a set of steps to be deleted and the strategy removes all steps that are computed by this function as well as all steps that depend on these.

For instance, the strategy BackTrackStepToTask in Fig. 4 instantiates the **BackTrack** algorithm with the function *stepback-to-task*. When BackTrackStepToTask is applied, it deletes all steps computed so far and takes the task (again) as the current task. The applicability condition of BackTrackStepToTask requests that the current task is not the initial conjecture.

5.2. Meta-reasoning with strategic control rules

Explicit control rules have been successfully used in PRODIGY [52] and they are an important ingredient of knowledge-based proof planning as well. They are used in our context to code procedural mathematically knowledge in the sense of "how to solve it". Technically, a control rule consists of an IF- and a THEN-part, where the IF-part is a proposition that describes features of proof plan states, the planning history and the current theory. The THEN-part modifies or just restricts the list of alternatives at the choice point for goals and methods. These control rules are marked as "tactic".

Strategic control rules are marked as "strategic" since they control the choice of a strategy. An example is `delay-ComputeInstCosie` whose declarative representation is given in Fig. 5. The IF-part checks whether there are still open goals (expressed by the predicate `goal-tasks`) and whether there are potential applications of the strategy ComputeInstFromCS in Fig. 3, which is expressed by the condition `job-offer`. If this is the case, the THEN-part rejects these applications of ComputeInstFromCS. Intuitively, the strategic control rule means: *as long as there are* *open goals, do not apply the strategy* ComputeInstFromCS, as this would lead to too many instantiations and hence a search space that is too bushy.

An advantage of this kind of explicit strategic control rules is that they are easily modifiable. For instance, the control rule `delay-ComputeInstCosie` can be removed or replaced by any other control rule. By contrast, if they were hard-coded, as it is often the case in a planner, each extension or change requires a re-implementation of the main control procedure. Moreover, the strategic control rules in MULTI can be ordered (currently by the user, but potentially by "knowledge sources" or agents) and they are evaluated in this order. A rule of thumb is that the rules are ordered by generality: specific strategic control rules are listed first and the more general ones later.

Blackboard systems separate the control of a knowledge source from its algorithm such that the control of the algorithm becomes an independent reasoning task. This led to hierarchical blackboards with several layers of control where one layer deals with the control of the next layer below. There is no such meta-meta level in our system as there appeared to be no need for it in our case studies. However, the system could easily be extended in this direction, if the need arises.

How sensitive is the MULTI system with respect to the strategic rule ordering? Indeed, the order matters. In the case studies reported below we worked with the same fixed ordering for all theorems of a domain and the ordering was not specifically tuned but followed the general principle of "specific rules before the more general ones". This worked so far for all our case studies but it should be subject to further research.

5.3. The MULTI *system*

The cooperation of the components of the MULTI system for proof plan refinement and modification is handled by a blackboard architecture, see [18]. In order to solve a problem, the components, called knowledge sources in the literature on blackboard systems, place the current solution state on the blackboard, which all knowledge sources can access. The activation of a knowledge source is determined by the data on the blackboard (created by other knowledge sources) and by extra control knowledge. In sophisticated blackboard systems this control is a first-class citizen. For instance, the HEARSAY-III [19] and the BB1 [24] architectures employ two separate blackboards: one blackboard to reason about the problem and one blackboard to reason about the control, i.e., the decision which of the applicable knowledge sources to apply next. So does the MULTI system whose control is not fixed a priori. The application of its knowledge sources is triggered by events and controlled via meta-reasoning.

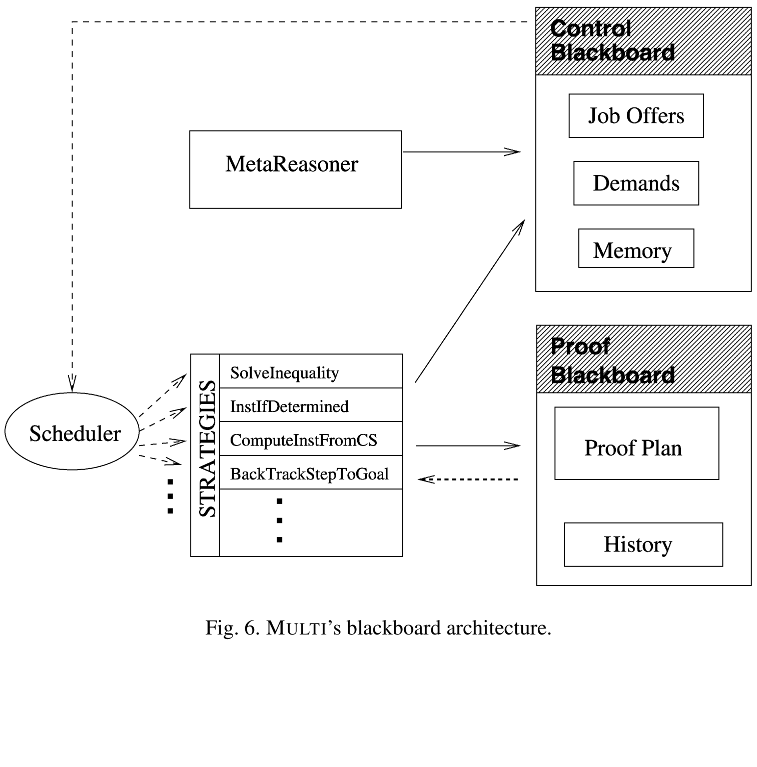

Fig. 6. MULTI's blackboard architecture.

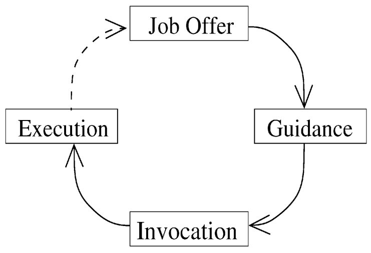

Fig. 7. MULTI's main stages.

5.3.1. *The architecture*

The system architecture is shown in Fig. 6. Dashed arrows indicate control flow whereas solid arrows indicate data flow, which changes the content of a repository on the blackboard.

The architecture consists of two blackboards: one for the current state of the proof, the *proof blackboard*, and one for the control problem, the *control blackboard*. The proof blackboard contains the current proof plan and its planning history and the strategies are the knowledge sources working on the proof blackboard. The control blackboard contains three repositories to store and organize *job offers*, *demands*, and *memory entries* to be explained below. The MetaReasoner is one of the knowledge sources working on the control blackboard. It evaluates the strategic control rules in order to rank the job offers. A `Scheduler` checks the control blackboard for its highest ranked job offer and executes the corresponding strategy.

A strategy posts its applicability as a *job offer* onto the control blackboard whenever the current situation satisfies the strategy's condition. A job offer is a pair consisting of a strategy and a task (the task, i.e. the goal the strategy is supposed to work on).

Whereas job offers are standard concepts in blackboard systems,[5] we use two additional repositories, called *demands* and *memory* to support an interrupt. A strategy may interrupt its processing if it gets stuck or if it needs another strategy to run before it can continue. If this is the case, the strategy interrupts its algorithm, places appropriate demands onto the control blackboard and stores its execution status in the *memory* for later continuation. The demand could include to prove a lemma or to compute some additional value. Typically, the strategic control prefers job offers responding to a demand. The strategy posts a job offer for its later re-invocation onto the control blackboard. When this job offer is scheduled later on, the interrupted strategy continues with the continuation information from the *memory*.

The overall mechanism follows the cycle in Fig. 7:

Job Offers Applicable strategies post their applicability as a job offer onto the control blackboard.
Guidance The MetaReasoner evaluates the strategic control rules and ranks the job offers.
Invocation The scheduler calls the strategy with the highest ranked job offer.
Execution The strategy is executed and it places its results on the proof blackboard. An interrupt or a termination can result in new demands and a new memory entry on the control blackboard.

This cycle is the well-known control loop in many expert systems and also in other blackboard architectures.

5.3.2. *Default strategic control*

The actual sequence of refinements and modifications of the proof plan is determined by the MetaReasoner, which evaluates the strategic control rules. While there is no pre-defined problem solving behavior in general, it turned out to be useful to have a default control regime nevertheless. The default control is operationalized in the three strategic control rules which manage *demands* and *memory* namely `prefer-demand-satisfying-offers`, `prefer-memory-offers`, and `defer-memory-offers` and in the strategic control rules, `prefer-backtrack-if-failure` and `reject-applied-offers`.

[5] Job offers are also called "knowledge-source activation records" (KSAR) in some systems.

- The rule `prefer-demand-satisfying-offers` suggests to prefer a job offer, if its job satisfies a demand on the control blackboard.
- Similarly, `prefer-memory-offers` suggests to prefer a job offer, if it is a job offer from an interrupted strategy in *memory* and all demands of this strategy execution are now satisfied.
- `defer-memory-offers` defers job offers from an interrupted strategy to the memory, if it has still unsatisfied demands.
- The purpose of the `prefer-backtrack-if-failure` rule is to combine goal-directed backtracking with the strategies of **PPlanner**. When a **PPlanner** strategy fails, it interrupts and stores the status of its current execution in *memory*. The strategic control rule `prefer-backtrack-if-failure` suggests to backtrack by assigning a high priority to a job offer of the BackTrackStepToTask strategy for the failing goal.
- The idea of the control rule `reject-applied-offers` is that a strategy, which failed on a task should not be tried again on this task. Hence, the rule rejects job offers from strategies that have unsuccessfully been applied to the task before.

As we shall see in the case studies below, more specific strategic control rules can overwrite these default rules if necessary.

5.3.3. Discussion of the architecture

A blackboard architecture is a far cry from the simple architectures of current automated reasoning systems, but then, why not use a more flexible and sophisticated control regime as for example in a multi-agent system. Since a strategy is proactive anyway, it could easily be encapsulated into an agent and the overall search behavior would then result from the structure and negotiation among strategy agents.

This would be particularly appropriate, if all control knowledge were local to the strategies. In case of mathematical problem solving and theorem proving, however, this is not the case—at least not in general. While there is plenty of local knowledge that pertains to the highly specialized tricks of the trade in every mathematical subfield, there is a great deal of general knowledge on how to find a proof and general proof techniques to guide the search. Hence, Alan Schoenfeld suggests [46, pp. 134–135] that

... it is useful to think of resources[6] *and control as two qualitatively different, though deeply intertwined, aspects of mathematical behavior.*

Therefore, we decided for a blackboard architecture with two separate boards that reflects the distinction between domain reasoning and control reasoning. The specialized knowledge (e.g. how to rearrange brackets, how to deal with inequalities, etc.) is only stored in the local methods and the tactic control rules.

Of course unstructured negotiation among a large set of agents is hardly ever used in the practice of multi agent systems and structuring devices such as holonic multi-agent systems or institutions have been invented to overcome these problems. In this case the architectural borderline between blackboard and multi-agent systems becomes hazy and it could be seen either way.

6. Evaluation

For the evaluation we have taken problems from several mathematical domains including ϵ–δ-problems, residues classes, permutation groups homomorphism from several text books in mathematics. Some of the ϵ–δ-problems were posed as challenges by Woody Bledsoe in the late 1980s [5] and most of these challenge theorems can still not be solved by current automated theorem proving systems. And those which can, such as `Lim+`, require extensive use of user-provided lemmata and control settings in order to come up with a proof. We have not only solved all of Woody's challenges, but have been able to prove many more ϵ–δ-theorems taken from [4]: all ϵ–δ-proofs in this standard text book are, in fact, now within the reach of our technology. Another large class of problems is taken from the

[6] Schoenfeld views the resources of a particular domain as: (1) informal and intuitive knowledge about the domain, (2) facts, definitions, and the like, (3) algorithmic procedures, (4) routine procedures, (5) relevant competencies, (6) knowledge about the rules of discourse in the domain (see [46, pp. 54–55]).

domain of residue classes. Further experiments are reported elsewhere and include permutation group problems [11], homomorphism problems [42] and the theorem that "$\sqrt{2}$ is irrational" [49].

At the end of this section we compare standard proof planning, i.e. the old ΩMEGA proof planner, with the multi-strategy approach.

6.1. Proof planning ϵ–δ-theorems

The domain-specific strategies we have used to tackle ϵ–δ-problems are the **PPlanner** strategy SolveInequality and the **InstVar** strategy ComputeInstFromCS from Section 2, which are complemented by the domain-independent **BackTrack** strategy BackTrackStepToTask. Using the default strategic control described in Section 5.3.2 and the strategic control rule `delay-ComputeInstCosie` from Section 5.2 the MULTI system proceeds as follows by default:

- All goals with (in)equalities or goals that can be decomposed to (in)equalities are handled by the method SolveInequality.
- When all goals are closed, then ComputeInstFromCS calls the constraint solver COSIE and instantiates the variables.
- If no method is applicable to a goal, then BackTrackStepToTask invokes backtracking.

Essentially, this default approach corresponds to the old proof planning and suffices to solve problems such as `Lim+`, `LIM*` (the theorem that the limit of the product of two functions equals the product of their limits) and many others that can be solved by classical proof planners (but are well beyond the reach of classical search-based systems). Proof planning with multiple strategies can solve much harder challenges, and in the remainder of this section, we shall discuss how this default approach is modified.

6.1.1. Eager instantiation

The theorem `LimPlusConst` introduced in Section 3 shows that the usual delay of the variable instantiation until the end is not always appropriate. A more suitable control would instantiate a variable, as soon as sufficient knowledge of how to instantiate is available. This can be expressed in the general meta-reasoning rule (that will be instantiated to the concrete situation at hand):

Eager Variable Instantiation:	
IF	there is knowledge of how to instantiate a variable
THEN	instantiate variable

How can we make this rule technical?

There are different sources for a suitable instantiation and different proof plan situations may require different actions to find a good instance. For example, external systems such as a computer algebra system, a constraint solver or a classical search based prover, could be called on demand to provide a value for the variable. In order to solve ϵ–δ-problems the system exploits the constraint solver COSIE, which handles restrictions of variables on reals. In this case the **InstVar** strategy InstIfDetermined encapsulates the necessary functionality of COSIE and the *Eager Variable Instantiation* rule becomes now:

IF the value for a variable is already uniquely determined by the currently collected constraints in COSIE, THEN instantiate the variable.

More concretely, the process works as follows: the cooperation between InstIfDetermined and SolveInequality is controlled by the rule `eager-instantiate`, which is part of SolveInequality. It guides the interrupt choice point of the **PPlanner** algorithm and fires, when COSIE sends a trigger that a variable has now a unique value. This interrupts SolveInequality and triggers the strategy to place a demand for InstIfDetermined on the *demand* repository on the control blackboard. After the instantiation of the variable by InstIfDetermined the system re-invokes SolveInequality.

Proof planning the theorem `LimPlusConst` illustrates this cooperation of the strategies. As described in Section 3, standard proof planning fails to prove the goals

$$|v_{x_1} - c| > 0, \tag{17}$$

$$|v_{x_1} - c| < c_{\delta_1}. \tag{18}$$

When these two goals are tackled by SolveInequality the constraints in COSIE already uniquely determine the value of the variable v_{x_1} as $c_x + c$. Hence, the strategic control `eager-instantiate` fires, interrupts SolveInequality and triggers the placing of a demand on the control board for InstIfDetermined. After the application of InstIfDetermined the system re-invokes the interrupted application of SolveInequality from memory. With the instantiation of v_{x_1} the two goals (17) and (18) become

$$\left|(c_x + c) - c\right| > 0, \tag{19}$$
$$\left|(c_x + c) - c\right| < c_{\delta_1}. \tag{20}$$

Now, a method calls an arithmetic simplifier which simplifies these goals to

$$|c_x| > 0, \tag{21}$$
$$|c_x| < c_{\delta_1}, \tag{22}$$

which then follow immediately from the given assumptions.

Other ϵ–δ-problems are similar and can be solved in the same way (see Section 6.1.3). Some of the residue class conjectures in Section 6.2 require eager variable instantiation too, but the value is computed by other external systems.

6.1.2. Failure reasoning

The failure to prove the `ContIfDerivative` theorem in Section 3 shows that a proof planning process can get blocked and the impasse may occur either inside a strategy or when choosing a strategy. In the first case, when a goal cannot be closed by a **PPlanner** strategy since no method is applicable, the failure is recorded and the strategy is interrupted as discussed above. In the second case, if no strategy is applicable, the failure is recorded too and in both cases, strategic control rules are now invoked to reason about the failure.

Some automated reasoning systems have a default behavior (for example critics) that is called at an impasse. So has MULTI. Its default behavior is encoded in the strategic control rule `prefer-backtrack-if-failure` which guides the application of the strategy BackTrackStepToTask. However, this default reaction *is only one of a variety* of possible reaction. Failure reasoning is not hard wired and fixed in the system, but coded into strategic control rules: the analysis of frequent failures and possible reactions to these led to a number of strategic rules, which are triggered by the failure situation and suggest suitable proof plan modifications or refinements. The following are two failure-related meta-reasoning rules for ϵ–δ-proofs. For more failure reasoning see [34].

Case split introduction

Some methods introduce side goals, called ‘conditions’. If one of these conditions cannot be proved (although the main goal is solved), this impasse can sometimes be resolved by a case split on the failed condition and the main goal has to be shown again for each of the two cases, i.e. first under the assumption that the condition holds and second that its negation is true.

It would *not* be appropriate to introduce a case split unconditionally for each open goal C as this would blow up the search space. Hence, this modification has to be controlled and the meta-reasoning rule that captures this intuition is:

Case Split Introduction:	
IF	failing condition C while some method M could solve main goal
THEN	introduce a case split in which C and $\neg C$ are assumed before application of M

The proof of the theorem `ContIfDerivative` provides an example to show how this strategic control rule can be used. As described in Section 3 proof planning initially fails to prove the condition $|c_x - a| > 0$. This failure triggers a case split before the method SolveInequality reduces the main goal $|f(c_x) - f(a)| < c_\epsilon$. Now SolveInequality has to prove the goal $|f(c_x) - f(a)| < c_\epsilon$ twice, in the first case by assuming $|c_x - a| > 0$ and in the second case by assuming $\neg(|c_x - a| > 0)$. In the first case, SolveInequality proceeds as previously described in Section 3 and derives the condition $|c_x - a| > 0$, which is the assumption. The second case is proved differently: first, SolveInequality simplifies the hypothesis $\neg(|c_x - a| > 0)$ to $|c_x - a| \leqslant 0$ and hence $c_x = a$. Now this equation is used to simplify the goal $|f(c_x) - f(a)| < c_\epsilon$ to $0 < c_\epsilon$, which then follows from the assumption.

This pattern of failure reasoning is used in other mathematical domains as well, where the general pattern is the same, however the actual case split depends on the mathematical domain. Typical examples for domain-dependent case splits are:

general case split on C: $C, \neg C$,
case split with real numbers: $a > 0, a < 0$, and $a = 0$,
case split for natural numbers: $n = 1, n > 1$,
case split in set theory: $x \in S$, $x \notin S$.

Unblock desirable steps

More often than not, proofs exhibit a common pattern with a particular combination of proof steps. If one of these steps is blocked within such a pattern during proof search, the system may be able to analyze how to unblock it.

This can be formulated as follows:

Unblock Desirable Steps:	
IF	strategy (or method) S is desirable but blocked
THEN	perform steps to enable S

How do we know that a step is desirable? And which steps enable S? The following elaborates on an example in which S is the strategy ComputeInstFromCS. Consider the theorem `LimDiv`, which states that the limit of the function $\frac{1}{x}$ at point $x = c$ is $\frac{1}{c}$, where $c \neq 0$. More formally:

$$\forall \epsilon \left(0 < \epsilon \Rightarrow \exists \delta \left(0 < \delta \wedge \forall x \left(x \neq 0 \wedge \delta > |x - c| > 0 \Rightarrow \left|\frac{1}{x} - \frac{1}{c}\right| < \epsilon\right)\right)\right).$$

`LimDiv` is a particularly hard problem for automated theorem proving and so far had not been solved by any system.

An important method to tackle this problem is FACTORIALESTIMATE. It is applied to inequality goals of the form $|\frac{t}{t'}| < e$. The method postulates the existence of a positive real number v and creates three simpler goals: $0 < v$, $|t'| \geqslant v$, and $|t| < e * v$.

Proof planning the `LimDiv` theorem works as follows: SolveInequality is applied to the theorem and the decomposition of the initial goal results in two new goals $0 < v_\delta$ and $|\frac{1}{c_x} - \frac{1}{c}| < c_\epsilon$. SolveInequality closes the first goal with TELLCS and simplifies the second goal to $|\frac{c - c_x}{c_x * c}| < c_\epsilon$. Now the system continues with the application of FACTORIALESTIMATE, which reduces this goal to three simpler subgoals $0 < v$, $|c_x * c| \geqslant v$, and $|c - c_x| < v * c_\epsilon$ with a new (auxiliary) variable v. These three goals can be closed by the constraint solver TELLCS. Since all goals are closed now, the strategy ComputeInstFromCS becomes highly desirable and should provide instances for the variables v_δ and v. These should be computed by the constraint solver but COSIE fails to compute a unique value for v_δ and v because so far the collected constraints are:

$\frac{\lvert c_x - c \rvert}{c_\epsilon} < v$	$0 < v$	$v \leqslant \lvert c_x * c \rvert$
$0 < v_\delta$	$c \neq 0$	$0 < c_\epsilon$

Hence, the application of ComputeInstFromCS is blocked.

To overcome this problem the system could tighten the constraints by further proof planning until a unique value can be derived from the constraint store. This technique is implemented in the strategic control rule `unblock-constraints`, which works as follows. If all goals are closed and there is no unique value from the current constraint store, `unblock-constraints` is invoked to analyze the current proof plan for further constraints. This is done by tracking all previous applications of TELLCS in order to derive tighter constraints.

More concretely, for the `LimDiv` theorem, the strategy `unblock-constraints` traces the applications of TELLCS that closed the goals $|c - c_x| < v * c_\epsilon$ and $|c_x * c| \geqslant v$. The strategy SolveInequality now reduces the re-opened goals even further with the method COMPLEXESTIMATE and passes the resulting constraints to the constraint solver COSIE. This leads to the constraint store (the variables v_1 and v_2 are introduced by the application of COMPLEXESTIMATE):

$c_\epsilon > 0$	$c \neq 0$	$v \geqslant v_1 * v_\delta$	$v_1 > c$
$v > 0$	$v_2 > 1$	$\frac{c_\epsilon * v}{2} > 0$	$v_\delta > 0$
$v_\delta \leqslant \frac{c_\epsilon * v}{2 * v_2}$	$v * 2 \leqslant c^2$		

Now the following instantiations can be computed from this set of constraints: $\{v_2 \mapsto 2,\ v_1 \mapsto c + 1,\ v \mapsto \frac{c^2}{2}$, and $v_\delta \mapsto \min(\frac{c_\epsilon * c^2}{8}, \frac{c^2}{2*(c+1)})\}$, which solves the problem.

This is not just an isolated example but many ϵ–δ-proofs require this kind of failure reasoning. Moreover, the control rule `unblock-constraints` is applicable in other domains as well, where constraint solvers are used.

These cases basically cover the essence of proof planning ϵ–δ-theorems, and the reader may wonder, how many tricks of the trade had to be collected. In other words, how much mathematical knowledge acquisition is required to cover a small mathematical subfield such as this?

All in all, we isolated and formulated about thirty methods (including very basic ones) and about half a dozen strategies for the domain of ϵ–δ-proofs. A core set of theorems can be proven just with the strategies SolveInequality and UnwrapAss. However, these methods were insufficient to solve theorems involving limits of fractions and other special cases, which required further methods to finally prove the large class of ϵ–δ-theorems in [4]. The main engineering effort is to detect a general pattern and then to capture it in the appropriate data structures of a method or a strategy. Once this work is done, further subclasses of theorems were provable without problems. We estimate that this effort for operationalizing mathematical thought and methods may take about twice as much time and resources as it takes a human novice to understand and master a mathematical subfield such as this.

6.1.3. Empirical results

Our strategies and methods are as general and generic as possible and all theorems within a given theory context (such as ϵ–δ-theorems) were run with the same setting of methods and strategies. The control rules were ordered from general to specific for each of the domains and remained fixed throughout, until we encountered a problem, the system could not solve. Then a new method, a new control rule or possibly even a whole new strategy had to be found, implemented and tested. This mirrors the situation of a maths student who improves her skills as she learns—in our case: acquire new methods and strategies. However after a while, this set had to be fixed and all problems were re-run and proved again with the new setting.

The system proved a large number of ϵ–δ-theorems (about seventy) from our main source, the textbook on analysis [4] and more similar problems could be formulated and proven. In fact, it is our claim that these theorems are now routinely within the reach of automated proof planning and that our current set of methods, strategies and control rules is essentially sufficient for this domain. This is an achievement which appeared to be nowhere on the horizon, say, a decade ago.

Now how can we empirically test and evaluate our final results?

A comparison with traditional automated theorem proving systems is not possible because even relatively simple ϵ–δ-theorems let alone the more difficult theorems we have been able to show, cannot be solved by any of these systems. Hence a comparison with previous (monolithic) proof planning appears to be more illuminating.

Table 1 presents several examples of ϵ–δ-proofs whose solution requires the flexible instantiation of variables or an explicit failure reasoning. The columns ((i), (ii), (iii) and (iv)) denote

(i) case split introduction
(ii) unblock constraint solving
(iii) lemma speculation and variable dependencies analysis
(iv) flexible variable instantiation.

A discussion of case split introduction (i), constraint solving (ii) and flexible variable instantiation (iv) has been presented in the previous paragraphs. Lemma speculation (iii) is a well-known technique in inductive theorem proving, where the speculated lemma ensures the applicability of the induction hypothesis. We are using this technique to overcome the failure of a desirable method application by the speculation of a lemma, which enables the application of this “desired” method.

Table 1
Typical ϵ–δ-proofs whose solution require flexible variable instantiation or meta-reasoning about failures. The column (i) records case split, (ii) unblock constraint solving, (iii) lemma speculation or variable analysis and (iv) records flexible variable instantiation

Conjecture	(i)	(ii)	(iii)	(iv)
$\lim_{x\to 0}(f(a+x)-f(a))=0 \Rightarrow cont(f,a)$	x		x	x
$\lim_{x\to a^-} f(x)=l \wedge \lim_{x\to a^+} f(x)=l \Rightarrow \lim_{x\to a} f(x)=l$	x			
$\lim_{x\to a^-} f(x)=f(a) \wedge \lim_{x\to a^+} f(x)=f(a) \Rightarrow cont(f,a)$	x			
$\lim_{x\to 2} \frac{1}{1-x}=-1$		x		
$\lim_{x\to c} f(x)=l_f \wedge \lim_{x\to c} g(x)=l_g \wedge \forall x\ g(x)\neq 0 \Rightarrow \lim_{x\to c} \frac{f(x)}{g(x)}=\frac{l_f}{l_g}$		x		
$\lim_{x\to\infty} f(x)=l \Rightarrow \lim_{x\to\infty} \frac{f(x)}{x}=0$		x		
$\lim_{x\to 0} f(x+a)=l \Rightarrow \lim_{x\to a} f(x)=l$			x	
$\lim_{x\to 0^+} f(\frac{1}{x})=l \Rightarrow \lim_{x\to\infty} f(x)=l$			x	
$\lim_{x\to 0} f(x)=l \wedge a>0 \Rightarrow \lim_{x\to 0} f(a*x)=l$			x	x
$\lim_{x\to c} f(x)=l \Rightarrow \lim_{x\to 0} f(x+c)=l$				x

Another technique analyzes variable dependencies and focuses on those variables that occur in several subgoals. If one goal cannot be proved, this may be caused by constraints on the shared variables. Hence, instead of backtracking, the pattern suggests to solve another already closed goal in a different manner.

Actually, 30 out of the 70 theorems involve a flexible instantiation of variables or failure reasoning and hence, previous proof planning systems cannot solve any of these. Also the high percentage of problems that require strategic reasoning demonstrates the *crucial role of strategic knowledge* in this domain.

When both, MULTI and previous proof planning (i.e. our old ΩMEGA system or λ-clam) can solve a problem, the two approaches do not differ much with respect to runtime nor wrt. the traversed search space. The reason is that strategic reasoning, i.e., the overhead of proof planning with multiple strategies, is minimal for this class of problems. The situation is quite different, however, for non-theorems, as the failure reasoning can produce substantial overhead. For instance, consider the non-theorem

$$\lim_{x\to a^-} f(x)=l \quad \Rightarrow \quad \lim_{x\to a} f(x)=l,$$

which is similar to the second problem in Table 1. When guided by failure reasoning, our system introduces a case split for this non-theorem, just as the case split necessary to actually solve the second problem in Table 1. This considerably enlarges the search space because of the failure point and the system does not just backtrack but explores far more alternatives.

6.2. *Proof planning residue class theorems*

This case study illustrates

- how mathematically motivated proof techniques can be coded into strategies,
- the flexible combination and cooperation of these strategies,
- the knowledge-based orchestration of services from external systems and
- meta-reasoning about the appropriate choice of a strategy which exploits knowledge of the performance and reliability of a strategy, knowledge from failure analysis and statistical knowledge.

There are three **PPlanner** strategies used throughout this large case study, namely TryAndError, EquSolve, and ReduceToSpecial. The strategy TryAndError makes an exhaustive case analysis, which is possible since the residue class theorems are about finite domains. In contrast, the strategy EquSolve avoids this exhaustive case analysis by reducing the assumptions and the conclusion to equations and then proves the goals by equational reasoning. ReduceToSpecial uses well-known simpler theorems about special cases from the database such as "*Two structures of different cardinality can not be isomorphic*".

Further strategies model typical human strategies for proving theorems about residue classes such as the discriminant technique shown below. And finally some strategies call external systems for cooperation:

- the **InstVar** strategy ComputeInstbyCasAndMG, calls the computer algebra systems MAPLE and GAP [21] as well as the model generator SEM [60] to compute suitable values for variables,
- the **InstVar** strategy ComputeDiscriminantbyHR calls HR to provide an instantiation for a meta-variable that satisfies some required properties,[7]
- the strategy NotInjNotIso models the (human) mathematical technique to infer a contradiction by assuming that there exists an isomorphism between two structures and then shows that this mapping is not injective,
- the **ATP** strategy CallTramp calls an appropriate external automated theorem prover. More technically, an ATP strategy calls an external prover via the math-bus MathServe, see [61] for details of MathServe. The ATP strategy CallTramp calls the Tramp mediator system [31] which then runs several provers such as Otter [29], Bliksem [14] and SPASS [53] concurrently.

6.2.1. *Residue classes*

To refresh our maths: what is a congruence? Take a number, say 3, and now observe the following: when 5 is divided by 3, you get 1 and the remainder is 2. Now take the number 8 and divide it by 3, then you get 2, but the same remainder as before, namely 2. We write: $8 \equiv 2 \bmod 3$ and say '8 is congruent 2 modulo 3'. The interesting thing is that the numbers that yield the same remainder modulo n can be put into a partition called a residue class, and we can compute with these classes as if they were single numbers. The congruence class 2 mod 3 is denoted as $\bar{2}_3$. Typical binary operations on residue classes are $\bar{+}$, $\bar{*}$, $\bar{-}$ which denote addition, multiplication and subtraction on residue classes. For example, $\bar{2}_3 \bar{+} \bar{1}_3 = \bar{3}_3 = \bar{0}_3$. These observations are centuries old and they are now part of *number theory*, where students learn to show basic algebraic properties for residue classes, e.g., that a residue class structure is a semi-group, a ring or a group. This part of number theory has recently found new and intensive interest because of its relevance to cryptography, where these properties, in particular the group property, are crucial for cracking (better: not being able to crack) a code.

A set of residue classes over the integers is either the set of all congruence classes modulo an integer n, i.e., $\mathbb{Z}_n$, or a subset of $\mathbb{Z}_n$, for instance, $\mathbb{Z}_3 \backslash \{\bar{1}_3\}$, $\{\bar{1}_6, \bar{3}_6, \bar{5}_6\}, \ldots$, where $\mathbb{Z}_3 \backslash \{\bar{1}_3\}$ denotes the set $\mathbb{Z}_3$ except the residue class $\bar{1}_3$.

$Iso(X, Y)$ means X and Y are isomorphic; $Hom(X, Y)$ means there is a homomorphism from X to Y; $Inj(h, X, Y)$ means that the mapping h from X to Y is injective and $Surj(h, X, Y)$ means that the mapping h from X to Y is surjective.

The first kind of problems we have looked at are conjectures about basic algebraic properties that are used to classify a given residue class structure. For instance, consider the set $\mathbb{Z}_5$ with multiplication $\bar{*}$, i.e., $(\mathbb{Z}_5, \bar{*})$. In order to classify the structure $(\mathbb{Z}_5, \bar{*})$ as a monoid we have to show that $\mathbb{Z}_5$ is closed under multiplication $\bar{*}$, that it is associative and that it has a unit element e. The property that for every element there is an inverse does not hold, hence the structure is not a group. These conjectures can be formalized as:

- $closed(\mathbb{Z}_5, \bar{*}) \equiv \forall x{:}\mathbb{Z}_5 \forall y{:}\mathbb{Z}_5 (x \bar{*} y) \in \mathbb{Z}_5$,
- $assoc(\mathbb{Z}_5, \bar{*}) \equiv \forall x{:}\mathbb{Z}_5 \forall y{:}\mathbb{Z}_5 \forall z{:}\mathbb{Z}_5 x \bar{*} (y \bar{*} z) = (x \bar{*} y) \bar{*} z$,
- $unit(\mathbb{Z}_5, \bar{*}, e) \equiv \exists e{:}\mathbb{Z}_5 \forall y{:}\mathbb{Z}_5 (y \bar{*} e = y) \wedge (e \bar{*} y = y)$,
- $\neg inverse(\mathbb{Z}_5, \bar{*}, \bar{e}_5) \equiv \neg \forall x{:}\mathbb{Z}_5 \exists y{:}\mathbb{Z}_5 (x \bar{*} y = \bar{e}_5) \wedge (y \bar{*} x = \bar{e}_5)$.

Other theorems are concerned with isomorphism properties of two given residue class structures, i.e., to prove or to refute that there exists a bijective homomorphism h. For instance, the structures $(\mathbb{Z}_5, \bar{+})$ and $(\mathbb{Z}_5, (x \bar{+} y) \bar{+} \bar{1}_5))$ are isomorphic, where the operation $op = ((x \bar{+} y) \bar{+} \bar{1}_5)$ in the second structure denotes the operation that adds two residue classes x and y, i.e., $x \bar{+} y$ and then adds $\bar{1}_5$ on top of it. By contrast, the structures $(\mathbb{Z}_5, \bar{+})$ and $(\mathbb{Z}_5, \bar{*})$ are not isomorphic. A formalization of the first conjecture is:

$$Iso\big((\mathbb{Z}_5, \bar{+}), \big(\mathbb{Z}_5, (x \bar{+} y) \bar{+} \bar{1}_5\big)\big)$$

and this can be restated as

$$\exists h{:}_{F(\mathbb{Z}_5,\mathbb{Z}_5)} Inj\big(h, (\mathbb{Z}_5, op)\big) \wedge Surj\big(h, \mathbb{Z}_5, (\mathbb{Z}_5, op)\big) \wedge Hom\big(h, (\mathbb{Z}_5, \bar{+}), (\mathbb{Z}_5, op)\big).$$

[7] HR is a system in the spirit of Doug Lenat's AM, which conjectures mathematical theories given empirical data. It was developed by Simon Colton [12] under the supervision of Alan Bundy and won the BCS/CPHC distinguished dissertation award as well as the best paper award at AAAI 2000.

The second conjecture can be formalized as:

$$\neg Iso\big((\mathbb{Z}_5, \bar{+}), (\mathbb{Z}_5, \bar{*})\big)$$

which can be restated as

$$\neg\exists h{:}_{F(\mathbb{Z}_5,\mathbb{Z}_5)} Inj(h, \mathbb{Z}_5) \wedge Surj(h, \mathbb{Z}_5, \mathbb{Z}_5) \wedge Hom\big(h, (\mathbb{Z}_5, \bar{+}), (\mathbb{Z}_5, \bar{*})\big).$$

6.2.2. *Proving basic algebraic properties for residue classes*

The three **PPlanner** strategies mirror general mathematical proof techniques in this context, namely:

(1) exhaustive case analysis by TryAndError,
(2) equational reasoning, by EquSolve, and
(3) the application of special theorems stored in our mathematical database, which are retrieved and then applied by the strategy ReduceToSpecial.

Both TryAndError and EquSolve cooperate with the **InstVar** strategy ComputeInstbyCasAndMG, which computes a value (an instantiation) for a variable calling a computer algebra system (CAS) or a model generator (MG). Once we have a value, the instantiation is 'eager' as discussed for ϵ–δ-proofs in Section 6.1.1.

TryAndError is the most reliable albeit the most obvious and slowest strategy. It creates large proof plans, whose size depends on the cardinality of the residue class. The proof plans of EquSolve are independent of the size of the residue class and, therefore, are in general much smaller than the proof plans of TryAndError. However, the strategy EquSolve fails for problems, which cannot be reduced to equations or which result in equations that cannot be solved by the computer algebra systems. ReduceToSpecial is also independent of the size of the residue class set. If it succeeds, it provides the most compact proof plans, but the success depends—as is to be expected—on whether suitable theorems are in the database.

Therefore, the MULTI system first employs fast but not always successful strategies and if they fail, the planner employs slower but more reliable strategies. This general principle is implemented in the strategic control rule `fast-before-reliable`, which orders the strategies we have so far in the order ReduceToSpecial, EquSolve, TryAndError.

6.2.3. *Isomorphism theorems*

The theorems in this class state that two given residue classes are isomorphic. They are proof planned with the strategies TryAndError, EquSolve, and ReduceToSpecial from above. First the three strategies are used just sequentially. If this succeeds, we are done. Otherwise the strategies cooperate, and we shall now show how this works.

The general idea of this cooperation is to switch between **PPlanner** strategies rather than backtrack, in case of failure. This makes sense, as backtracking may erase valuable progress, while other promising strategies are available that can use this information and continue with the problematic goal. Technically, this idea is encoded in the strategic control rule `preferotherjob-if-failure`. It fires only on 'important' goals, i.e., major subproblems (injectivity, surjectivity, homomorphism) that are part of the definition (in this case the definition of isomorphism).

The proof planning process which generates the most compact and interesting proof plan results from the following cooperation of strategies: First EquSolve is applied to the theorem, which calls ComputeInstbyCasAndMG for an eager instantiation of the variable. Then EquSolve solves the homomorphism and surjectivity subproblems. The resulting subproblem, namely to show injectivity cannot always be proved by EquSolve. Thus, guided by the strategic control rule `preferotherjob-if-failure` the system applies the strategy ReduceToSpecial. This strategy solves an injectivity goal by applying a theorem, for example, the following: "*A surjective mapping between two finite sets with the same cardinality is injective*", which the system has proved before and stored in the database.

6.2.4. *Non-isomorphism problems*

Non-isomorphism problems can also be solved by the three strategies TryAndError, EquSolve, and ReduceToSpecial. However more interesting are two other mathematically motivated proof techniques for non-isomorphism theorems:

(1) If two structures are isomorphic, then they share all algebraic properties. Thus, in order to show that two structures are not isomorphic it suffices to find a property that holds for one structure but not for the other. Such a

property is called a *discriminant*. This general mathematical technique is realized by a combination of the **PPlanner** strategy EquSolve, an HR strategy to find discriminants, an **ATP** strategy, and **InstVar**.

(2) Proof by contradiction: assume there exists an isomorphism between two structures and then show that it is not injective. This technique is modeled by the strategy NotInjNotIso.

The strategic control rule `fast-before-reliable` ranks NotInjNotIso after EquSolve and before TryAndError since it is (usually) less efficient than the application of EquSolve but more efficient than TryAndError.

In the following, we outline the two techniques and their application to residue class problems.

Discriminants This technique for proving that two structures S^1 and S^2 are not isomorphic consists of the following steps: first show the general theorem

(1) $\forall X \forall Y$ if $P(X)$ and $\neg P(Y)$ then $X \not\sim Y$ (where X and Y are variables for structures and P is some property) and store it in the database.
Then:
(2) find a discriminant P,
(3) show that $P(S^1)$ holds,
(4) show that $\neg P(S^2)$ holds.

The strategy EquSolve introduces the goals (3) and (4) and a meta-variable for the discriminant P. Next, EquSolve interrupts and waits for an instantiation for P, which is computed by the strategy ComputeDiscriminantbyHR. The **InstVar** strategy ComputeDiscriminantbyHR calls the system HR and if it succeeds, the general theorem (1) has to be proven. In order to do so, several first-order automated theorem provers are concurrently run by the **ATP** strategy CallTramp. All of this is still domain-independent but the proof plans for (3) and (4), i.e. $P(S^1)$ and $\neg P(S^2)$, are domain-specific and they are generated by the **PPlanner** strategies TryAndError, EquSolve, and ReduceToSpecial.

The orchestration of all these services works demand-driven: to prove (3) and (4) first, the strategy EquSolve is applied, then it interrupts and triggers a demand. This demand is satisfied by the **InstVar** strategy ComputeDiscriminantbyHR and then the **ATP** strategy CallTramp to prove the goal, which was generated by HR. When both strategies succeed, EquSolve is re-invoked and it tackles the goals (3) and (4) with respect to the discriminant. If EquSolve fails to prove these subgoals, then ReduceToSpecial and TryAndError are applied and guided by the strategic control rule `preferotherjob-if-failure`. More applications of this *discrimination technique* are discussed in [37].

Proof by contradiction Given two structures S^1 and S^2, NotInjNotIso constructs a proof by contradiction. First the strategy assumes that there exists an isomorphism $h : S^1 \to S^2$ and then tries to find two elements $c_1, c_2 \in S^1$ such that $c_1 \neq c_2$ and $h(c_1) = h(c_2)$. This contradicts the postulated injectivity of h, where $h(c_1) \neq h(c_2)$ would follow from $c_1 \neq c_2$. In order to show $h(c_1) = h(c_2)$, NotInjNotIso starts with this goal and then computes equality substitutions until it obtains an equational goal, which can be solved by the computer algebra systemMAPLE.

The strategy NotInjNotIso can produce very short proofs even for structures with very large sets. However, choosing c_1 and c_2 and constructing an appropriate sequence of equality substitutions is the hard part, which may not always terminate.

To overcome this dilemma we experimented with randomization and restart techniques known from [22]. The basic idea is to cutoff and restart a randomized algorithm, possibly many times, on a given problem rather than to let it run for a long time. How and when to cutoff and how often to restart is derived from statistical knowledge extracted in previous experiments [30].

To exploit this technique in multi-strategy proof planning we added control rules to NotInjNotIso that randomly select c_1 and c_2 as well as the equational substitutions. The cutoff and restart option is captured in two control rules: `interrupt-if-cutoff` interrupts NotInjNotIso, when the run time exceeds a predefined cutoff and poses a demand to backtrack with the **BackTrack** strategy BackTrackPPlannerStrategy. The strategic control rule `reject-applied-offers` (described in Section 5.3.2) prevents NotInjNotIso to be applied again to the same initial goal. However, `reject-applied-offers` can be overwritten by the more specific strategic control rule `restart-`NotInjNotIso, which now allows the application of NotInjNotIso several times. More examples and empirical results can be found in [33].

Table 2
Results of the experiments on residue class problems

	Simple properties				Iso-classes		
	All	$\mathbb{Z}_5$	$\mathbb{Z}_6$	$\mathbb{Z}_{10}$	$\mathbb{Z}_5$	$\mathbb{Z}_6$	$\mathbb{Z}_{10}$
Magmas	8567	3049	4152	743	36	7	14
Abelian magmas	244	53	73	24	26	5	6
Semi-groups	2102	161	1114	35	3	8	1
Abelian semi-groups	2100	592	1025	62	1	12	2
Quasi-groups	1891	971	738	70	9	2	10
Abelian quasi-groups	536	207	257	11	3	2	1
Abelian monoids	211	97	50	6	1	1	1
Abelian groups	1001	276	419	49	1	1	1
Total	18,963	5406	8128	1000	80	38	36

6.2.5. *Empirical results*

We constructed a testbed of a large number of residue classes (about 20,000). Their cardinality range from 2 to 10 and their binary operations were systematically constructed from the basic operations $\bar{+}$, $\bar{-}$, $\bar{*}$. These structures were then classified in terms of their algebraic categories and many theorems were proved in order to identify different isomorphism classes. Table 2 presents some results of this classification. Altogether, we have classified 18,963 structures with respect to their algebraic properties, which required to proof plan about 60,000 theorems.

To test the validity of the above techniques for isomorphism as well as non-isomorphism (Sections 6.2.3 and 6.2.4) the system proved properties of the structures $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$. We identified about 160 isomorphism classes, which required to proof plan about 1300 non-isomorphism theorems and about 2000 isomorphism theorems.

The system successfully employed ReduceToSpecial for about 20%, EquSolve for 23% of the proofs; the remaining 57% of the theorems were solved by the TryAndError strategy. 88% of the isomorphism proof plans were constructed by a cooperation of EquSolve, ReduceToSpecial and TryAndError, 12% of the isomorphic theorems were shown by TryAndError alone. For the non-isomorphism problems 18% of the proof plans were found through a discriminant; the remaining 82% with the NotInjNotIso strategy.

There was not a single case for which the combined proof techniques realized in the **PPlanner** strategies failed. This success is entirely due to proof planning with multiple strategies and their cooperation.

Many of the theorems in this class are still in the range of traditional automated theorem proving systems. Thus, the challenge and our motivation for these experiments was not only their difficulty—although most examples are pretty hard—but the mathematically interesting and different proofs that could be generated. Our system was tested against the first-order prover WALDMEISTER [25], which is particularly well-tuned for equational theorem proving. Moreover, the WALDMEISTER settings and the formalization of the theorems were highly tuned by a WALDMEISTER expert on the basis of system-specific and mathematical knowledge. Two different control settings were used, one suitable for non-isomorphism problems and one for all other problems. Under these highly specific circumstances, which allowed a special tuning for each theorem given to WALDMEISTER, MULTI performed comparably well. The proof planner, apart from being more general of course, produced proofs that were mathematically better structured and closer to a human representation. In particular it could prove some of the hard problems, which WALDMEISTER could not.

WALDMEISTER has a clear advantage over proof planning with respect to runtime behavior: when it succeeds, then typically in less than a second whereas problems take about 20 seconds with MULTI (independently of the cardinality of the residue class set). A disadvantage of WALDMEISTER is its output format: although this system employs an extra component for structuring and presenting proofs, they are usually very long and the structure of the proof is often mathematically counter-intuitive. There are usually between 150 and 300 equational reasoning steps, structured with 10 to 30 lemmas and most of these lemmas do not make sense mathematically but are necessary for the system to succeed. In contrast, proof planning produces short and comprehensible proof plans: the plans generated by ReduceToSpecial or EquSolve have about 10–20 steps. Proof plans generated by TryAndError can be considerably longer but these proof plans are still more clearly structured by their case splits.

This case study shows that although many of these theorems are still within the range of traditional search-based automated theorem provers[8]—albeit highly tuned and special purpose for equational reasoning—the difference to proof planning is striking: the planned proofs are very natural mathematically, easy to read and to comprehend and significantly (sometimes by one order of magnitude) shorter.

Summary The above case studies as well as the experience with other empirical work we have done so far shows that everyday mathematical theorem proving can indeed be done with a computer, provided we have enough common sense mathematical knowledge coded into the system. Proof planning significantly pushes the limit

(i) with respect to the naturalness and human oriented nature of the generated proofs as well as
(ii) the overall strength of the system in terms of the difficulty of the successfully proven theorems.

6.3. *Extensibility, flexibility and maintenance of the architecture*

Other dimensions to evaluate multi-strategy proof planning versus standard proof planning systems are extensibility, flexibility and maintenance of the system.

When a proof planner can not prove a theorem X, say, because of an inappropriate instantiation of variables, the fix is to extend the current instantiation mechanism for meta-variables or to change the algorithm of a refinement or modification functionality such that the proof planner can now prove X (and all previously proven theorems).

For each new theorem that can not be shown, the code needs to be changed. After fixing the proof planner for X, we soon face the next problematic theorem $X + 1$ and so on: the code becomes more and more baroque and possibly interferes with other components too. This evolutionary approach is typical for a knowledge based systems. This is indeed what happened with our old ΩMEGA system. The modularity is now a major advantage as usage, updating and maintenance is greatly improved.

6.4. *Lessons learned*

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

(W. Bledsoe 1986)

While the results recorded in this paper significantly push the limits of what can be achieved by a machine today one doubt may remain:

How many tricks of the trade and how much of the highly specialized knowledge has been coded into the system?

For a classical search based automated theorem prover (frozen intellectually in a time warp of the 1960s and early 70s of AI technology) representing and using special tricks of the mathematical trade used to be considered a *vice*: "if you help the system that way, no wonder it can prove the theorem" (see [23] for this kind of discussion). For a knowledge-based system, however, it is the most important *virtue* to provide *general* means for the representation of and reasoning with domain-specific knowledge.

But how general is the knowledge we have coded into the system?

The basic methodology of strategic proof planning is *general purpose*: a *planner*, *methods* as operators, *control rules* and *strategic control rules* to guide the search, classical *automated theorem provers* and *computer algebra systems* as service providers, a *constraint solver* and a system for *mathematical theory invention* and finally, a *blackboard architecture* to orchestrate the overall reasoning—all of this is now general and standard technology in artificial intelligence.

[8] This is one of the reasons why we have chosen this data set of theorems, i.e. to make a comparison possible.

The domain knowledge however, which represents mathematical techniques and heuristic guidance, is—as the name suggests—specific for the field under scrutiny. But it should not be too specific for just a few problems. Instead it must be general enough to solve a *whole class* of problems (see also [8] for a general discussion of this issue).

So, how can we be convinced that we have not coded too many specific tricks into the strategies, methods and control rules? After all, any solved problem, no matter how hard it is, can be solved by any other automated system given enough time to tune the dials for a replay.

A viable criterion is the amount of coverage and the range of theorems in the mathematical field. In particular, routine proofs with similar proof ideas should be provable with the same strategies.

We claim that after extensive experimentation the coded knowledge is general and sufficient for proving the whole class of theorems within the investigated fields. Consider, e.g., the SolveInequality strategy, which encodes mathematical knowledge to solve inequality problems such as ϵ–δ-theorems. The SolveInequality strategy is an instance of a more general heuristic problem solving strategy described by Alan Schoenfeld:

In a problem 'to find' or 'to construct', it may be useful to assume that you have the solution to the given problem. With the solution (hypothetically) in hand, determine the properties it must have. Once you know what those properties are, you can find the object you seek. [46] (p. 23)

Could we encode a more general version of this strategy "assume, collect properties, then compute" that subsumes SolveInequality? Well, not to the best of our knowledge. Even for human mathematical problem solving, Schoenfeld points out that such a general heuristic strategy alone does not provide enough information for solving a concrete problem:

[...] *a typical heuristic strategy is very broadly defined—too broadly, in fact, for the description of the strategy to serve as a useful guide to its implementation.* [46] (p. 70)

Rather, such general strategies have to be "filled-in" with domain-specific knowledge and the general strategy is only a summary label for a *class of strategies* suitable for different domains:

[...] *the successful implementation of heuristic strategies in any particular domain often depends heavily on the possession of specific subject matter knowledge.* [46] (p. 92)

Thus SolveInequality can be seen as an instantiation of "assume, collect properties, then compute" for the domain of inequality problems for real numbers. SolveInequality and its used *methods* and *control rules* turned out to be sufficient for the whole class of ϵ–δ-proofs, i.e., a class of theorems first year maths students spend several months to learn.

In summary the system development and its test on several large sets of typical theorems in a particular mathematical field demonstrate:

- a *knowledge-based* automated reasoning system—as envisaged by Woody Bledsoe in the 1980s in [6]—can in fact be built with general purpose techniques.
- Once we have accumulated enough domain specific knowledge, the system can perform on par with any human expert in that particular domain.

But, as we know well, any professional mathematician can definitively do more than just prove the theorems of a well established mathematical field, so we will not run out of research problems soon.

7. Related work

The ΩMEGA system and its MULTI planner have many characteristics in common with other blackboard systems, as discussed in Sections 5.3 and 6.4. In particular, unblocking desirable steps is a general pattern of control reasoning in other blackboard systems [13,17]: when a highly desirable knowledge source is not applicable, then reasoning on the failure can suggest the invocation of knowledge sources that unblock the desired knowledge source. Despite this general reasoning pattern, however, the concrete implementation in strategic control rules has to rely on domain-specific knowledge.

7.1. Combination of algorithms and systems

Competitive combinations of planning systems [20,26] as well as systems in automated theorem proving [47,51, 59] are motivated by the fact that there is no single system that outperforms all other systems on all problems. Hence, a combined system applies several systems or several parameterizations of one system to the same problem and terminates as soon as one system succeeds.

Although multi-strategy proof planning has also been used for the competitive combination of strategies (e.g., to solve residue class problems the system calls several **PPlanner** strategies) an interesting feature is the *cooperation* of strategies. The MULTI planner shares this kind of cooperation with the multi-agent approach of TECHS [15] and the planning architecture MPA [57]. TECHS realizes the cooperation of several automated theorem proving systems by wrapping them with communication facilities such that they become an "agent". Each agent then uses so-called *referees*, which decide which information of its search state should be communicated to which of the other agents and which information received from the other agents should be integrated into its own search state.

In MPA, planning-related technologies as well as management functionalities are also wrapped with communication facilities. For instance, MPA [58] has plan expansion agents and plan critics agents for causal link protection and for constraint consistency. A scheduler agent provides services and handles the allocation of resources.

7.2. Proof planning

Proof planning in CLAM

The proof planner CLAM [9] and its successor λCLAM [44] were developed by Alan Bundy's group in Edinburgh. Both systems specialize mainly on proofs by mathematical induction and the preconditions of their methods may include legal and heuristic conditions. Control knowledge is not separated from methods. In λCLAM a proof method can be atomic or compound, similar to the *tactics* in LCF. A compound method is called a "strategy" and it is constructed from simpler methods with pre-defined constructors called methodicals similar to *tacticals* [43]. For instance, the compound method `(THEN M1 M2)` first applies the method `M1` and afterwards `M2`. Andrew Ireland and Alan Bundy extend proof planning in CLAM by so-called *critics* as a means to patch failed proof attempts in proof planning inductive proofs [27,28]. Critics in CLAM are method like entities which are triggered by method failure and they can have global effects on the proof tree. More specifically, a critic is associated with a method and captures a patchable exception to the application of this method. A critic can, e.g., introduce a lemma that is needed by the `wave` method. In contrast, failure reasoning in MULTI is represented by control rules, which are not associated with a particular method. Rather, they reason about the current proof plan and the proof planning history. The patch of a failure is not implemented into special procedures but into methods and strategies whose application is invoked by general control rules. MULTI solves the job of lemma speculation and case analysis in a way that is independent and not hard wired into the particular method/critic.

The need for more flexibility was recognized for CLAM as well. For instance, a fixed order of the compound methods generalization, simplification, induction was found to disable some proofs. Therefore, a scoring mechanism was developed to determine the "most promising" expansion of the proof plan.[9] In contrast, MULTI's control knowledge describes situations and choices declaratively rather than selecting one of a small number of alternative orderings of strategies, our notion of a strategy is more general. In both systems control is heuristic and, thus, can fail so failure analysis remains a necessity.

Proof planning in IsaPlanner

The IsaPlanner [16] is the most recent proof planner developed in Edinburgh. It is linked to the interactive proof development system Isabelle. IsaPlanner has a functional approach to proof planning, where the functions are applied to a proof state consisting of a proof plan and a 'continuation'. IsaPlanner does not rely on pre-defined declarative methodicals that are evaluated by an interpreter. Similar to MULTI, IsaPlanner introduces more flexibility into the proof planning process. Moreover, IsaPlanner can now call a critic at any time via a methodical.

[9] Unpublished work.

8. Conclusion and future work

Proof planning with multiple strategies generalizes our previous, rather monolithic proof planning process of the ΩMEGA system. Different services are now encapsulated into strategies that can be flexibly combined and controlled by meta-reasoning. Strategies and their control constitute a new hierarchical level of the problem solving process above and more general than the current search for a proof plan. This allows us to separate and code mathematical knowledge into methods, control rules, strategies and strategic control.

Proof planning with multiple strategies also provides a convenient framework to integrate the user. First results on mixed-initiative proof planning are reported in [36] and island planning in [38], where initial 'islands', i.e., the important "Heureka Steps" are provided by the user and the system works out the tedious rest of the proof. These techniques were also successfully used for the "challenge" theorem stating that $\sqrt{2}$ is irrational (see [49]).

Future work

One difference of the current Ωmega system—as compared to the state reported in this paper—concerns the logical representation of proofs. The system enjoys now a a more high-level logical representation at the assertion-level, which makes some low-level reasoning obsolete [1].

One future direction is the extension of proof planning to new classes of theorems. A first step in this direction has been made by our colleagues Volker Sorge and Martin Pollet together with the mathematician Arjeh Cohen who applied the described techniques to Permutation Group problems [11].

Another direction of our current and future work is to accumulate more (standard) proof techniques and implement these in appropriate strategies. Proofs by induction are highly mechanized today and we are just using one of these systems when there is the need. Another standard technique is diagonalization, which was developed for proof planning in ΩMEGA several years ago and we are now re-implementing it in the new framework. Proofs by analogy are (again) worked on and implemented within the notion of a context.

An interesting source book is Daniel Solow: "How to Read and do Proofs", which collects a few dozen proof techniques for teaching and we are currently implementing these for proof planning.

Current proof planning (even with domain-specific, i.e., contextual strategies and control) is still in stark contrast to the situation of a mathematician who operates in a specific *context*, this includes the current theory under development, preferences, knowledge representation techniques tailored to the specific context, definitions that are formulated in a way to serve the current purpose, typical techniques to prove this particular theorem as well as tricks of the trade typical for the field within which the theorem is stated. For example, a theorem about a continuous function requires certain strategies to tackle its proof, which a student learns in the calculus courses, whereas a theorem, say in group theory, requires a very different set of strategies usually taught in an algebra class. The choice of either of them *is prior and above* strategic reasoning and determined by the current context. Of course, a deep human spark of mathematical insight and creativity may correspond to a switch to another context, as happens more often than not in mathematics, e.g., in the proof of Fermat's last theorem. But, alas, we are not there (yet) with computer-supported mathematics.

Another goal are non-theorems that has been tried but is not yet fully implemented. The idea is that strategic control rules can call a model generator to find a counter-example to an open goal. If a counter-example is found, a strategic control rule can interrupt the current strategy and cause backtracking to the open goal. The strategic control rules could, e.g., use a data base of typical counter examples or typical non-theorems of a domain and check them systematically against the open goal.

Acknowledgement

The work reported in this paper was done in a time span of several years and this paper reports the state of development up to and including the year 2004. Many members of the ΩMEGA-group contributed directly or indirectly in more ways we can possibly remember: This lively environment made this work possible and we like to thank them all. More specifically, we like to thank Chris Benzmüller, Serge Autexier and, in particular Martin, Pollet, Volker Sorge and Jürgen Zimmer for their help with the case studies.

The work was funded by the DFG-project MIPPA (Me 1136/2-1) and the collaborative research center (Sonderforschungsbereich 378, Resource Adaptive Cognitive Processes) of the German National Science Foundation (DFG). We are grateful for this substantial funding that made this work possible.

References

[1] S. Autexier, Hierarchical contextual reasoning, Computer Science Department, Saarland University, Saarbrücken, Germany, PhD thesis, 2003.
[2] O. Gandalf, CASC-14, http://www.cs.jcu.edu.au/~tptp/casc-14/, 1997.
[3] F. Baader (Ed.), Proceedings of the 19th International Conference on Automated Deduction (CADE-19), Miami Beach, FL, LNAI, vol. 2741, Springer-Verlag, Germany, 2003.
[4] R. Bartle, D. Sherbert, Introduction to Real Analysis, John Wiley & Sons, New York, 1982.
[5] W. Bledsoe, Challenge problems in elementary analysis, Journal of Automated Reasoning 6 (1990) 341–359.
[6] W. Bledsoe, I had a dream: AAAI presidential address, AI Magazine August (1985) 57–61.
[7] A. Bundy, The use of explicit plans to guide inductive proofs, in: E. Lusk, R. Overbeek (Eds.), Proceedings of the 9th International Conference on Automated Deduction (CADE-9), Argonne, IL, in: LNCS, vol. 310, Springer Verlag, Germany, 1988, pp. 111–120.
[8] A. Bundy, A critique of proof planning, in: Festschrift in Honour of Robert Kowalski, 2002.
[9] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, Experiments with proof plans for induction, Journal of Automated Reasoning 7 (1991) 303–324.
[10] A. Church, A formulation of the simple theory of types, Journal of Symbolic Logic 5 (1940) 56–68.
[11] A. Cohen, S. Murray, M. Pollet, V. Sorge, Certifying solutions to permutation group problems, in: [3], 2003.
[12] S. Colton, Automated Theory Formation in Pure Mathematics, Springer-Verlag, London, 2002.
[13] D. Corkill, V. Lesser, Hudlicka, Unifying data-directed and goal-directed control, in: D. Waltz (Ed.), Proceedings of the Second National Conference on Artificial Intelligence (AAAI-82), Carnegie-Mellon University/University of Pittsburgh, Pittsburgh, PA, AAAI Press, Menlo Park, CA, 1982, pp. 143–147.
[14] H. De Nivelle, Bliksem User Manual, Delft University of Technology, 1998.
[15] J. Denzinger, D. Fuchs, Cooperation of heterogeneous provers, in: T. Dean (Ed.), Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI), Stockholm, Sweden, Morgan Kaufmann, San Mateo, CA, 1999, pp. 10–15.
[16] L. Dixon, J. Fleuriot, IsaPlanner: A prototype proof planner in Isabelle, in: [3], 2003, pp. 279–283.
[17] E. Durfee, V. Lesser, Incremental planning to control a blackboard-based problem solver, in: T. Kehler, S. Rosenschein (Eds.), Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, AAAI Press, Menlo Park, CA, 1986, pp. 58–64.
[18] R. Engelmore, T. Morgan (Eds.), Blackboard Systems, Addison-Wesley, 1988.
[19] L. Erman, P. London, S. Fickas, The design and an example use of HEARSAY-III, in: B. Buchanan (Ed.), Proceedings of the 6th International Joint Conference on Artificial Intelligence (ICJAI), Morgan Kaufmann, Tokyo, Japan, 1979, pp. 409–415.
[20] E. Fink, How to solve it automatically: Selection among problem-solving methods, in: [50], 1998, pp. 128–136.
[21] GAP, GAP—Groups, Algorithms, and Programming, Version 4, The GAP Group, 1998.
[22] C. Gomes, B. Selman, H. Kautz, J. Mostow (Eds.), Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) and Tenth Conference on Innovative Application of Artificial Intelligence (IAAI-98), Madison, WI, AAAI Press, Menlo Park, CA, 1998, pp. 431–437.
[23] P. Hayes, D.B. Anderson, An arraignment of theorem-proving; or, the logician's folly, Memo 54, Dept. Computational Logic, Edinburgh University, 1972.
[24] B. Hayes-Roth, P. Hayes, A blackboard architecture for control, Artificial Intelligence 25 (1985) 251–321.
[25] T. Hillenbrand, A. Jaeger, B. Löchner, System description: WALDMEISTER, improvements in performance and ease of use, in: H. Ganzinger (Ed.), Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, in: LNAI, vol. 1632, Springer-Verlag, Germany, 1999, pp. 232–236.
[26] A. Howe, E. Dahlman, C. Hansen, M. Scheetz, A. von Mayrhauser, Exploiting competitive planner performance, in: S. Biundo, M. Fox (Eds.), Recent Advances in AI Planning, Proceedings of the 5th European Conference on Planning (ECP'99), Durham, UK, in: LNCS, vol. 1809, Springer Verlag, Germany, 1999, pp. 62–72.
[27] A. Ireland, The use of planning critics in mechanizing inductive proofs, in: A. Voronkov (Ed.), Proceedings of the 3rd International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg, Russia, in: LNAI, vol. 624, Springer-Verlag, Germany, 1992, pp. 178–189.
[28] A. Ireland, A. Bundy, Productive use of failure in inductive proof, Journal of Automated Reasoning 16 (1–2) (1996) 79–111.
[29] W.W. McCune, Otter 2.0 users guide, Argonne National Laboratory, ANL-90/9, 1990.
[30] A. Meier, Randomization and heavy-tailed behavior in proof planning, Seki Report SR-00-03, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2000.
[31] A. Meier, TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level, in: D. McAllester (Ed.), Proceedings of the 17th Conference on Automated Deduction (CADE-17), in: LNAI, vol. 1831, Springer-Verlag, Germany, 2000, pp. 460–464.
[32] A. Meier, Proof planning with multiple strategies, Ph.D. thesis, University of Saarland, FB Informatik, Saarbrücken, Germany, 2004.
[33] A. Meier, C. Gomes, E. Melis, Randomization and restarts in proof planning, in: A. Cesta, D. Borrajo (Eds.), 6th European Conference on Planning (ECP-01), in: LNCS, Springer, 2001.
[34] A. Meier, E. Melis, Impasse-driven reasoning in proof planning, in: Proceedings of Fourth International Conference on Mathematical Knowledge Management (MKM2005), in: M. Kohlhase (Ed.), LNAI, vol. 3863, Springer-Verlag, 2005, pp. 143–158.

[35] A. Meier, E. Melis, MULTI: A multi-strategy proof planner, in: R. Nieuwenhuis (Ed.), Automated Deduction: 20th International Conference on Automated Deduction (CADE-20), Tallinn, Estonia, in: LNAI, vol. 3632, Springer-Verlag, 2005, pp. 250–254.

[36] A. Meier, E. Melis, M. Pollet, Adaptable mixed-initiative proof planning for educational interaction, Electronic Notes in Theoretical Computer Science 103 (C) (2004) 105–120.

[37] A. Meier, V. Sorge, S. Colton, Employing theory formation to guide proof planning, in: J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, V. Sorge (Eds.), Proceedings of Joint International Conferences, AISC 2002 and Calculemus 2002, Marseille, France, Springer-Verlag, Germany, 2002, pp. 275–289.

[38] E. Melis, Island planning and refinement, Seki Report SR-96-10, Universität des Saarlandes, FB Informatik, 1996.

[39] E. Melis, AI-techniques in proof planning, in: H. Prade (Ed.), Proceedings of the 13th European Conference on Artificial Intelligence, John Wiley & Sons, Brighton, UK, 1998, pp. 494–498.

[40] E. Melis, U. Leron, A proof presentation suitable for teaching proofs, in: S.P. Lajoie, M. Vivet (Eds.), 9th International Conference on Artificial Intelligence in Education, IOS Press, 1999, pp. 483–490.

[41] E. Melis, J. Siekmann, Knowledge-based proof planning, Artificial Intelligence 115 (1) (1999) 65–105.

[42] M. Pollet, E. Melis, A. Meier, User interface for adaptive suggestions for interactive proof, in: Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP 2003), Rome, Italy, 2003.

[43] J. Richardson, A. Smaill, Continuations of proof strategies, in: R. Gore, A. Leitsch, T. Nipkov (Eds.), Short Papers of International Joint Conference on Automated Reasoning, 2001.

[44] J. Richardson, A. Smaill, I. Green, System description: Proof planning in higher-order logic with λ*Clam*, in: C. Kirchner, H. Kirchner (Eds.), Proceedings of the 15th International Conference on Automated Deduction (CADE–15), Lindau, Germany, in: LNAI, vol. 1421, Springer-Verlag, Germany, 1998, pp. 129–133.

[45] A. Robinson, A. Voronkov, Handbook of Automated Reasoning, vol. 1, Elsevier, 2001.

[46] A. Schoenfeld, Mathematical Problem Solving, Academic Press, New York, 1985.

[47] J. Schumann, SiCoTHEO—simple competitive parallel theorem provers based on SETHEO, in: Proceedings of PPAI'95, Montreal, Canada, 1995.

[48] J. Siekmann, C. Benzmüller, S. Autexier, Computer supported mathematics with ΩMEGA, Journal of Applied Logic (2006).

[49] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, M. Pollet, Proof development in OMEGA: The irrationality of square root of 2, in: F. Kamareddine (Ed.), Thirty Five Years of Automating Mathematics, in: Kluwer Applied Logic series, Kluwer Academic Publishers, 2003.

[50] R. Simmons, M. Veloso, S. Smith (Eds.), Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98), Pittsburgh, PEN, AAAI Press, Menlo Park, CA, 1998.

[51] G. Sutcliffe, D. Seyfang, Smart selective competition parallelism ATP, in: A. Kumar, I. Russell (Eds.), Proceedings of the 12th International Florida Artificial Intelligence Research Symposium (FLAIRS-99), Orlando, Florida, USA, 1999, pp. 341–345.

[52] M. Veloso, J. Carbonell, M. Perez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: The prodigy architecture, Journal of Experimental and Theoretical Artificial Intelligence 7 (1) (1995) 81–120.

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

[54] G. Weiss (Ed.), Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, MIT Press, Cambridge, MA, 1999.

[55] D. Weld, An introduction to least commitment planning, AI Magazine 15 (4) (1994) 27–61.

[56] D. Wilkins, M. desJardins, A call for knowledge-based planning, Artificial Intelligence 22 (2001).

[57] D. Wilkins, K. Myers, A multiagent planning architecture, in: [50], 1998, pp. 154–162.

[58] D. Wilkins, K. Myers, M. desJardins, P. Berry, Multiagent planning architecture, Tech. rep., Stanford Research Institute (SRI), 1997.

[59] A. Wolf, Strategy selection for automated theorem proving, in: F. Giunchiglia (Ed.), Artificial Intelligence: Methodology, Systems and Applications, Proceedings of the 8th International Conference (AIMSA'98), Sozopol, Bulgaria, in: LNAI, vol. 1480, Springer-Verlag, Germany, 1998, pp. 452–465.

[60] J. Zhang, H. Zhang, SEM: A system for enumerating models, in: C. Mellish (Ed.), Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI), Montreal, Canada, Morgan Kaufmann, San Mateo, CA, 1995, pp. 298–303.

[61] J. Zimmer, S. Autexier, The mathserve system for semantic reasoning with web services, in: IJCAR, 2006.

[62] J. Zimmer, E. Melis, Constraint solving for proof planning, Journal of Automated Reasoning 33 (1) (2004) 51–88.