On the Comparison of Proof Planning Systems λCLAM, ΩMEGA and ISAPLANNER

Louise A. Dennis [1,2]
School of Computer Science and Information Technology
University of Nottingham, Nottingham, UK

Mateja Jamnik [1,3]
University of Cambridge Computer Laboratory
University of Cambridge, Cambridge, UK

Martin Pollet [1,4]
Fachbereich Informatik
Universität des Saarlandes, Saarbrücken, Germany

Abstract

We present a framework for describing proof planners. This framework is based around a decomposition of proof planners into planning states, proof language, proof plans, proof methods, proof revision, proof control and planning algorithms.

We use this framework to motivate the comparison of three recent proof planning systems, λCLAM, ΩMEGA and ISAPLANNER, and demonstrate how the framework allows us to discuss and illustrate both their similarities and differences in a consistent fashion. This analysis reveals that proof control and the use of contextual information in planning states are key areas in need of further investigation.

Key words: Proof Planning

1 Introduction

Proof planning was introduced by Alan Bundy [1] as a new paradigm for proof automation. Rather than using low level logical inference rules, the proof construction is automated using so-called *proof methods* which capture common patterns of reasoning.

[1] This work was supported by EPSRC grants GR/N37314/01 and GR/S01771/01, an EPSRC Advanced Research Fellowship GR/R76783, SFB 378 Project grant, and European Commission IHP Calculemus Project grant HPRN-CT-2000-00102. Thanks to Lucas Dixon for work on an earlier version of this paper and to Ewen McLean for fruitful discussions.

[2] Email: `lad@cs.nott.ac.uk`

[3] Email: `Mateja.Jamnik@cl.cam.ac.uk`

[4] Email: `pollet@ags.uni-sb.de`

Proof planners search for a *proof plan* of a theorem. This plan can then be executed to derive a fully formal proof in terms of the underlying logical inference rules. The search space tends to be smaller than that at the level of inference rules [3,8]. This is due to the more abstract nature of proof methods appearing in the plan and, in particular, the structuring of the search space which is made possible by this abstraction. Proof planning therefore focuses on providing mechanisms for organising the application of these proof steps. A salient feature of proof planning is that it generates proofs where the reasoning patterns are transparent, and where failure can be patched.

Recent case studies have shown that proof planning provides a basis for the integration of computational systems for both guiding the automatic proof construction and performing proof steps [7,4]. For example, computer algebra systems can be used to instantiate variables in proof steps, constraint solvers to collect and administrate inequalities, and theory formation systems to construct discriminating properties of algebraic structures.

These case studies show that proof planning is a suitable framework for the integration of computational algorithms into deduction systems, but a modern analysis of the proof planning approach (in particular, with reference to failure recovery) is still missing. We feel it is timely to consolidate the ideas, the progress and state-of-the-art in this field. Therefore, it is the objective of this paper to present (in §2) clear abstract definitions of the individual features of proof planning in order to provide a common *framework* for the discussion and comparison of proof planning systems. We illustrate the use of our framework by presenting (in §3) a comparison of three recent proof planning systems: λCLAM, ΩMEGA and ISAPLANNER.

2 A Framework for Proof Planning

Proof planning, as the word describes, is conceptually related to traditional planning in Artificial Intelligence (AI). Describing a problem in traditional AI planning consists of logical sentences about the initial state and the goal state. The goal state is described by a logical query sentence which asks how some situation can be achieved from the initial state. A proof problem can be viewed as a planning problem when we identify the local assumptions of the problem and the theory in which the problem is formulated as the initial state. The conclusion of the proof problem corresponds to the goal state.

The planning operators describe the transformation of planning states into new states. A planning operator is typically represented by its precondition and effects. The precondition specifies what needs to be true in the planning state for the operator to be applied. The effect of an operator specifies how the situation changes after the operator has been applied. The use of *proof methods* in proof planning is derived from this paradigm.

The (proof) planner searches for a sequence of operators with which the initial state can be transformed into the goal state. The sequence of operators of a successful process is a plan. Since there is no need to linearise methods, a *proof plan* is a tree or a DAG (directed acyclic graph).

A proof plan is similar to an AI plan in that it represents the description of how a planning problem (theorem) can be solved (proved), as opposed to representing the execution of the plan (formal proof). As in AI planning, the structuring of the planning process uses search algorithms and control knowledge to decide which path to take at choice points.

The usage of many terms in proof planning has become somewhat overloaded and confused. Where possible we have sought to retain common terminology, but we will endeavour to be precise about our usage. It is important to note that the terminology used in our framework is based on function and not on nomenclature. It is possible that some identically named feature may appear in two proof planning systems which our framework categorises differently for each system (e.g., the compound methods of λCLAM can only loosely be placed in the proof method category of our framework). We consider this to be a strength since it allows differences to be more easily perceived even when terminology is confusing.

Our framework is based on a classification into components most of which are available in all proof planning systems.

2.1 Proof Planning State

Most literature on proof planning describes proof planners as operating on partial proof plans. However, an important insight of Dixon and Fleuriot [5] is to notice that proof planners operate on *proof planning states*.

We consider all components of the state, that is, all *sources* of information *and* all parts that can be *manipulated* by proof planning. These are:

Proof goals: goals to be proved or solved.

Proof plan: a proof at some level of abstraction (can be partial).

History: a record of the proof search process, including backtracked steps and failed operators.

Control information: used to select the next proof planning operator.

Context: a repository to store additional information. This information is either attached to terms, to the proof plan, or is in the form of control status attached to goals (e.g., why a goal was not solvable).

The least well-defined part of the planning state is the context. It is different in all of the systems we considered. The context is useful for performing search for a proof of a problem, but it is not necessary for reconstructing a formal proof from the proof plan.

During proof planning, as in AI planning, the proof planning state is transformed by proof planning operators. It is possible to derive numerous categories of operators depending upon which bits of the planning state they can access and which they can manipulate. However, in practice the uses of planning operators in proof planning fall into distinct common categories:

Proof methods are descriptions how to transform goals to new subgoals; they may also use and manipulate information from the context.

Proof revisions describe the reaction to failure. They are a global operation on the goals, proof plan, control knowledge and context information.

Further categories of operator are associated with the following mechanisms:

Proof control is used to select the next goal and the next proof plan operator. Proof control may use proof control operators to manipulate control knowledge, which is then analysed (potentially along with the proof plan, history and context) to select the next operator.

Proof expansion is the transformation of the proof plan into a formal proof.

In the following we describe these operators and mechanisms in more detail. The proof expansion is considered in the context of the hierarchical structure of proof plans and the language used for formal proofs.

2.2 Proof Language

The proof language (object language) is the language of formulae and the inference rules of the logical calculus within which a proof planning attempt takes place. The proof on the object level is usually the final result of the proof planning process, this object level proof is generated from the proof plan.

Systems differ in the expressiveness of the object language, for example, some object languages exclude meta-variables (free variables introduced when searching for a witness to existentially quantified variables).

2.3 Proof Plans

The term *proof plan* was originally used to describe both an explicit structuring of the search space and a high level representation of the proof itself. In later descriptions the term has generally been used to describe the latter. Therefore we adopt this meaning: a proof plan describes the result of proof planning. We will use the term *proof control* to describe search space structuring.

A proof plan is a graph which stores the result of the application of proof methods to goals, and is the basis for the expansion of an abstract proof into a calculus level proof. A reference to the proof method is stored and can be seen as the justification for the node. Therefore, a proof plan can be interpreted as an abstract proof, or rather, a proof sketch for the problem. Proof plans may contain different, and possibly many, levels of abstraction.

The expansion of proof methods results in more detailed proof plans, with the formal proof at the lowest level. The construction of the formal proof can either be interleaved with proof planning steps, or implemented as a verification phase that comes after the proof search. There are a variety of techniques for expanding a proof plan down to a more concrete one. At present expansion happens locally for each single proof method in all systems. This means that the structure of the proof plan is determined by the logical calculus (see §3.2).

Abstraction is realised by the mechanisms that are used for proof control. Sequences of proof methods can be contracted to a single step which is labelled with the proof control responsible for the selection of this sequence of methods.

2.4 Proof Methods

A proof method is a description of how to transform a goal into new subgoals and also presents a justification for this manipulation. It has no access to the information held in the current proof plan or history, nor does it know about other methods. Only the particular goal and the context information relevant to that goal are available to the method.

In the literature, we find that proof methods are often defined as declarative descriptions of tactics, where a tactic is a function which can be applied to a goal to generate subgoals and which guarantees the correctness of the derivation. The declarative nature of a method does not necessarily provide any information about the manipulation performed by the method in advance of its application (see §3.4.1). In this respect, a tactic and declarative method contain the same degree of information, and the only difference is that a declarative method needs an interpreter to actually perform the manipulation.

Clearly, at some level, proof methods affect changes in the planning state, particularly the partial proof plan. The relationship between proof methods and proof planning operators varies from system to system.

The difference between tactics and methods is a source of much debate. Our framework allows us to observe that a tactic cannot access any contextual information so tactics form a subset of methods. Every method which acts solely on goals expressed in the proof language for formulae (i.e., on G to produce subgoals $\bar{S}$) can be emulated by a modus ponens-like tactic that introduces an additional subgoal for the implication (i.e., $\bar{S} \vdash G$). So a method distinguishes between subgoals required for solving the problem on an abstract level, and subgoals (in some systems left implicit) required for a formal verification. The primary distinguishing feature of methods with respect to tactics therefore, is that methods have access to and are able to manipulate context information, whereas LCF-style tactics only manipulate object level formulae. As a secondary feature, methods often leave some subgoals implicit or at least separately categorised to indicate they are to be solved by expansion, whereas a tactic makes all its subgoals explicit and gives them all the same status.

2.5 Proof Revision

Proof revision is the most powerful mechanism available in proof planning. Proof revision is triggered in all systems by failure. Failure usually means that a goal could not be closed, either because no proof method was applicable, or the planner looped. However, it is possible for failure to be interpreted more generally as any process that determines the need for some major change of the approach. The standard reaction to failure is backtracking: a proof plan operator that deletes nodes from the proof plan and updates control information appropriately to prevent that branch from being re-explored.

The abstract nature of proof planning allows very sophisticated analysis of failures. The analysis is either based on proof methods, or the proof plan and history. The specification of how to react includes information about where to start again (manipulation of the proof plan) and how to continue (manipulation of proof control information).

2.6 Proof Control

Proof control uses the control information in the planning state to determine the selection of the next goal and the proof planning operator.

In current systems, proof control information is either expressed in form of *control rules* which analyse the current proof plan and the history, or it is a description how proof plan operators should be sequenced and combined. This description is manipulated as proof planning proceeds and actual operators applied. The updated description is called the *continuation*. In some cases the continuation is itself treated as a proof plan operator.

Therefore, across the various systems proof control is dictated by a mixture of control information in the planning state and proof plan operators that modify this information. It is important to consider the control operators, the control information, and the mechanisms for using the information to select the next operator together in order to understand how the proof search is controlled. Therefore, in our framework we consider all these together under the heading of proof control rather than individually as part of the planning state, planning operators and planning algorithm.

2.7 Planning Algorithm

The final component of a proof planning system is the planning algorithm. The planning algorithm is responsible for performing the search for a proof plan that solves the given problem. It has to process control information, apply proof methods, update the proof plan, and switch to proof revision.

3 A Comparison of Three Proof Planning Systems

We now illustrate our framework by using it to compare proof planners. A summary of the main characteristics of these systems in relation to each other and with respect to our framework is given in Table 1 at the end of this section.

Proof planning is implemented in several systems. The first proof planner, CLAM [1], was designed to prove theorems by mathematical induction. Its successor, λCLAM [11], has been extended to higher order logic.

Another well known system is the ΩMEGA proof planner [9]. In ΩMEGA, proof planning is viewed in the tradition of human-oriented reasoning techniques for mathematical theorem proving (as opposed to logic-oriented techniques such as resolution). ΩMEGA integrates various other mathematical services, such as traditional automated theorem provers, a knowledge base, computer algebra systems, etc.

Finally, the IsaPlanner system [5] is a proof planner for the interactive theorem prover Isabelle [10]. IsaPlanner interleaves proof planning with the proof plans' execution and thus is able to take advantage of powerful tactics available in Isabelle.

We use a comparison of these three proof planning systems as a case study to illustrate how our framework from §2 can be used as a basis for discussing the construction of proof planning systems.

3.1 Proof Planning States

In IsaPlanner the planning state is a triple consisting of a proof plan, proof context, and control information in form of a continuation. Goals are implicitly represented as the open nodes of the proof plan. The proof context may contain arbitrary information. IsaPlanner has access to Isabelle, including theory libraries and information on rewrite rules to be used in simplification. A proof planning operator is a function on planning states and is called a *reasoning technique*.

λCLAM's planning state also consists of a proof plan and a continuation. The context information λCLAM uses includes annotations attached to terms and a number of global lists (such as theorems to be used for rewriting).

The planning state of ΩMEGA contains goals, the proof plan, control rules, one or more constraint stores as context, and the history of proof search. Goals can be proof goals, that is, formulae to be proved, or instantiation goals for variables. The constraint store is used to collect information for the instantiation goals. There exist additional constraint stores for different domains. The history has three parts: the proof plan viewed as a history of successful method applications, the history of backtracked steps and a strategic history. History is an essential part of ΩMEGA's proof planning state. It is used by the proof control and we will discuss this in depth in §3.6. ΩMEGA also makes use of annotations attached to terms and goals.

Our analysis of the proof planners revealed that some of the information used by planing operators was not explicitly mentioned as part of the planning state. For example, the global lists in λCLAM in which theorems are stored and which are used by proof methods for rewriting. Similar types of parameters are also present in ΩMEGA, but they are implicitly given within ΩMEGA's control rules. Ideally, such information should be made explicit as part of the planning state context, which would allow reasoning with them, for example, to start a new planning attempt with a modified list of theorems.

Since the different systems use different sorts of context information, and since this information is also dependent on the proof domain, we propose the concept of *proof context* as an extensible repository for information. In interactive theorem provers the "context information" needed to prove a theorem is provided by the human user, in proof planning this information has to be modelled and represented within the system.

The notion of context can cause complications. Since there are a number of different types of context information, operators may have to be aware of these distinctions. Mixing operators which depend and act on different types of context information, could lead to inconsistencies between the proof plan and the context. Further work to analyse and understand the role of proof context in proof planning is clearly needed.

3.2 Proof Language

ΩMEGA uses a typed λ-calculus for formulae. The base calculus is a higher-order version of Gentzen's Natural Deduction Calculus (ND).

λCLAM's object language is a sequent based typed λ-calculus. λCLAM's predecessors used a version of Martin-Löf constructive type theory, and λCLAM clearly owes much to these systems. However, λCLAM adopts no clear set of object level inference rules. As such, its object language is a syntactic entity with a loose interpretation as a higher-order sequent calculus.

ΩMEGA and λCLAM treat free variables resulting from existentially quantified variables as meta-variables which are not part of the proof language.

IsaPlanner uses Isabelle's language for formulae. This means it has access to a rich language that implements meta-variables at the object level.

3.3 Proof Plans

All three proof planning systems represent proof plans as DAGs. In the case of IsaPlanner and λCLAM, this representation is restricted to a tree. The nodes of these DAGs are labelled with goal formulae and hypotheses. In all systems the nodes contain some reference to the proof operation responsible for their creation which can be interpreted as an abstract justification of the node. ΩMEGA's use of DAGs is clearly a more general representation of the proof plan than the trees used by λCLAM and IsaPlanner. The DAGs in ΩMEGA allow nodes to be reused for the justification of several goals.

Proof plans may be expressed at different levels of abstraction. The transformation to more detailed proof plans is achieved by expansion. Expansion is used to generate a verifiable formal proof.

The verification of a proof plan in ΩMEGA is done in a second phase after the proof search: all meta-variables are replaced by their witness terms, and each proof method is expanded into a subproof which may contain other methods. The process stops when all proof steps are calculus level rules, so that the fully expanded proof can be checked.

The proof plans produced by λCLAM justify individual proof steps by the names of their corresponding proof methods. In λCLAM, there is no actual expansion of the proof plan available. Nevertheless, a mechanism similar to the expansion in Ωmega was envisioned. Each proof method is associated with a tactic (for an LCF-style theorem prover), and thus the proof plan can be viewed as a tactic tree which can be executed.

In IsaPlanner expansion is interleaved with the application of proof methods. A method is only applicable when a formal proof for the manipulation performed by the method can be constructed. The proof plan is represented as proof script in the Isar language.

In the comparison of the different systems a clear design choice emerges between whether proof verification is interleaved with the construction process or it is delayed. Postponing the expansion phase can allow proof search to use efficient algorithmic methods to calculate subgoals and leave actual calculation of the formal proof (which may be time-consuming) until it is certain that this branch of the plan will actually appear in the final version. It also allows a proof plan to be generated independently of any object language peculiarities. On the other hand, interleaved expansion allows the system to use, where appropriate, tactics that exist in the underlying system. In a two phase (i.e., postponed) expansion a failure in the expansion generally leads to a revision of the whole proof plan, whereas with an interleaved verification this kind of incorrect proof plan is ruled out during proof search. The interleaved execution is equivalent to a two phase approach where it is possible to mark goals in the interleaved execution as expansion goals.

The original idea behind proof planning was to construct a meta level proof plan and only later reconstruct the underlying formal proof. This was based on the observation of mathematical practice, where it is rarely necessary to worry about the use of precise logic in a "pen-and-paper" proof, but where such proofs *can* usually be formalised to a particular logic, if necessary. The local expansion of methods means that the formal proof is more detailed than the proof plan while it still preserves the same structure. This strong connection between the formal proof and the proof plan influences the implementation of proof operators. Some logics place more constraints on abstract proof plan construction than others. For instance, ND is very sensitive to the order in which proof steps are taken and this dictates the way some proof operators are implemented. However, from the "pen-and-paper" perspective, the order of proof steps should only depend on the problem, and not on the verification.

Proof plans can also be abstracted by folding up sequences of steps. This is primarily used in interfaces. It allows a user to "zoom in" or "zoom out" of parts of the DAG which may be difficult to comprehend in its entirety.

3.4 Proof Methods

We use the term proof method here for descriptions of how to transform a goal into a (possibly empty) list of subgoals, and which may also manipulate the context. These are referred to as atomic methods in λClAM and methods in Ωmega. They are not clearly identified in IsaPlanner, but are a subset of the reasoning techniques available in that system.

λClAM and Ωmega's proof methods are expressed in a frame data structure containing input, output, parameter and precondition slots. The input and output slots are used to match and generate goals. This frame structure provides an interface for the users to create their own methods, either using arbitrary λProlog code in λClAM, or a specialised interpreted language developed for this task in Ωmega.

IsaPlanner uses a single and extensible language for encoding techniques, which are functions on planning states. It is possible to identify a subset of these techniques as proof method-like in that their primary function is the production of new subgoals. Some of them even contain the declarative information similar to method preconditions in the proof context. This declarative information is used to enable proof revision.

In Ωmega and λClAM the (semi-)declarative structure for proof methods allows them to be viewed as descriptions of a proof step which the system subsequently applies to create a proof planning operator. The system automatically updates the proof plan and history etc. as appropriate. IsaPlanner's reasoning states are explicitly functions on proof planning states. Insofar as we identify some of these as methods, it is those which extend the partial proof plan with new goal nodes. It is also worth noting that some of IsaPlanner's reasoning techniques are created by applying wrappers to tactics which lift them from functions on goals to functions on proof planning states.

In most publications on proof planning, parameters are neglected in the description of methods. We explicitly mentioned parameters as slots in the frame data structure used for methods, because this allows an information flow from the proof control layer to proof methods. Parameters can be instantiated when a method is selected by the proof control.

3.4.1 Declarativity

Of the different formalisms for the encoding of methods, Ωmega's has the highest degree of declarativity. The language for the input and output is fixed, there exists an interpreted language for preconditions, and the expansion is represented by a proof schema which contains a declarative specification of the subproof. It was envisioned that the language for preconditions would reach a fixpoint, but it turned out that new domains need new types of preconditions. In λClAM the preconditions are directly expressed in the programming language. In contrast, the declaration of slots in IsaPlanner is not fixed, and declarativity is provided only by the optional inclusion of "precondition-like" statements in the proof context. So we cannot speak of fully declarative language for proof methods in any system.

There are three primary motivations for adopting a declarative language: the analysis of operators; easing the implementation for the system users who wish to create their own proof planning operators; and identifying how to introduce (and remove) steps.

The first of these reasons has not, in practice, been of much utility for proof planning. Many methods in proof planning systems cannot be analysed. For example, consider a method which applies a computer algebra algorithm to do simplifications on the goals. The only way to know the effect of this method is to apply it to a concrete goal.

Users often only need simple tools for syntactically manipulating formulae which can be provided using input schema. More sophisticated methods seem to require a fully flexible programming language in which to write arbitrarily complex preconditions which expert users will need to master. The case for a declarative language between these two levels of complexity appears thin.

Thus there remains only one reason for the use of declarative preconditions which is their use in analysing steps for expansion or during failure reasoning. Once again it seems likely that only the input and output schemes are really necessary here since other information can be re-constructed (see §3.5).

3.5 Proof Revision

λCLAM has explicit proof critics [6] which are used to great effect with rippling [2]. Critics are traditionally triggered when a method fails to apply, but may also be triggered proactively, and tend to use an analysis of the method's preconditions to motivate a revision of the proof plan. In contrast to methods, which only add new goals, a critic will often delete or replace nodes. IsaPlanner's proof planning operators can also perform this sort of large-scale manipulation of the proof plan. They use declarative information stored by methods in the proof context instead of explicit precondition analysis. In essence, critics are used to jump over branches of the search space which would otherwise be explored laboriously (typically by backtracking), and to introduce new areas into the search space.

In Ωmega there are two possible types of unsuccessful exits of a proof planning strategy. Either there is no applicable method (called *failure*), or the proof planning strategy itself can cause an *interruption.* The analysis of failures at the strategy layer is called *meta reasoning.* Meta reasoning is based on analysis of the current goal, the partial proof plan, and the history, rather than on a method's preconditions.

All systems allow detection of failures, which are not directly connected to one method, for example, loops in the proof search.

A critic which analyses a given method's preconditions has limitations. Since the language for preconditions is not totally declarative, a critic depends on how the precondition is expressed. Thus, the introduction of a critic can make it necessary to change the proof method,and a change in the proof method has consequences for the corresponding critic. This inter-dependency between critics and proof methods can be avoided when the critic performs its own analysis of the proof planning state, or the method stores additional information about the failure in the proof context.

A revision usually consists of several different parts, for example, backtracking, changing the proof control, continue on a specific goal. In ΩMEGA these steps are implemented as different strategies, but (strategic) control rules can only select the immediate next strategy. This means, it is not possible to express the exact combination of those steps. After each step it is necessary to analyse what should be performed next. For this it is necessary to differentiate between backtracking steps performed for different revisions so that the intended next step of the revision is selected.

3.6 Proof Control

In λCLAM and IsaPlanner, methodicals are used to combine methods into *methodical expressions*. The methodical expression $\texttt{then\_meth}(M_1, M_2)$ is interpreted as an application of first method M_1 and then method M_2, if M_1 was applicable. Methodicals express the structure of a proof, for example, $\texttt{then\_meth}(\texttt{rewrite}, \texttt{orelse\_meth}(\texttt{assumption}, \texttt{tautology}))$ controls a proof that will attempt to rewrite a goal and then prove it using either the proof method for assumptions or by showing the goal is a tautology. Methodicals express a plan for how the proof search should progress from this point. Practically they restrict the possible planning operators considered at any point in the proof search. Methodical expressions can be composed from both proof plan operators and other methodical expressions. This allows the structuring and reuse of both operators and these compound controls. Methodical expressions can also be used to abstract a proof plan: the subtree of methods which were introduced by a methodical expression can be contracted to an edge which has the methodical expression as justification. Methodical expressions are stored in planning states as a continuation, and the continuation is modified as planning progresses (e.g., in the above example, once the `rewrite` method was applied, the continuation is $\texttt{orelse\_meth}(\texttt{assumption}, \texttt{tautology})$. In λCLAM the user supplies an appropriate methodical expression in a fixed language at the start of planning, and the system interprets this as planning progresses. IsaPlanner treats methodicals as proof planning operators that can be programmed by users to work in a similar but completely extensible fashion to those used in λCLAM.

In ΩMEGA the selection of methods is controlled by *control rules*. Control rules exist for the different choice points of the planning algorithm: goals, proof methods, and actions (instantiated methods). At these choice points, the planner has to choose one object from a list of objects in a planning state in order to continue. The control rules consist of conditions and modify the list of objects given to it when the conditions holds. The control rules are executed on the initial list of objects which is filtered and reordered to form a new planning state. Later, the planner processes the list of objects in the given order. The condition part of a control rule analyses the proof plan, and the proof history. We can say that methodical expressions express the 'future' of a proof attempt. Whereas control rules look into the 'past' by analysing the history and the proof plan to make a decision about the next step.

In ΩMEGA, there is an additional layer for strategies. Strategies were introduced because it was inconvenient to have an unstructured set of methods and control rules. Strategies are collections of methods and control rules, and provide implicit proof structuring. One proof planning attempt may employ several strategies. This serves a conceptually similar purpose to the way in which a methodical expression can be composed of other expressions. The choice of strategy is controlled by strategic control rules. There are also control rules which can interrupt a strategy.

The language for expressing proof control in the form of methodicals is equivalent to tacticals in interactive theorem proving. A methodical expression expresses a search pattern. An instantiation of this pattern is the proof plan for a concrete theorem. The current set of methods and control rules embodied in a strategy clearly also encode a proof pattern, but less explicitly than is possible with methodicals. In particular the effects of modifying and adding control rules are hard to predict because control rules interact via the list of options they modify. Paradoxically, it seems that while adopting a more declarative approach for the construction of proof methods, ΩMEGA has used a less declarative approach to control knowledge.

We believe the approaches have the same expressiveness in the sense that control rules can emulate the interpretation of methodical expressions and control rules can be emulated by methodicals when there is a conditional methodical.[5] So the approaches are different only in the style in which proof control is expressed. For instance, it is possible to express that a simplification method should be tried after each method application with one control rule. To express the same behaviour with methodicals requires the addition of the simplification method to all proof methods in the methodical expression. On the other hand it is relatively cumbersome to express a sequence of proof methods with control rules, whereas this is straightforward with the methodical `then_meth`. A combination of both approaches that would exploit the respective benefits of the different styles constitutes further work.

3.6.1 Separation of Heuristic Knowledge

In ΩMEGA proof methods are expressed as generally as possible, while heuristic information is delegated to control rules. This means that a method may be

[5] Further work, to appear.

re-used in a variety of different situations by adapting these control rules. It also means that undesired method applications are rejected on the control level, rather than encoding this decision in the design of the method. The consequence of this separation is that for different situations different control rules have to be implemented for the application of the same method.

λCLAM and IsaPlanner do not make such distinctions clear. Potentially this can lead to one proof step having several different instantiations as proof methods with different sets of heuristic preconditions, depending on where it is to be used in methodical expressions. It is possible that including heuristics in methods makes the construction of methodical expressions simpler.

Placing heuristic knowledge in proof methods reduces reusability, but does appear to allow proof control to be more declarative. It is not obvious how to directly compare such differing approaches to the separation of heuristic and legal knowledge, and we plan further exploration of this area.

3.7 Planning Algorithm

Of the three systems under consideration Ωmega employs the most complicated planning algorithm, and the only one that is comparable to AI planning systems. Ωmega uses a depth-first search strategy with respect to the results of control rules, but the algorithm passes through several stages. First, relevant control rules are used to select a current goal for consideration. Then, they are used to select an appropriate method, the method is interpreted and tested on the selected goal. If it is applicable, then the proof plan, the constraint store and the history are updated and the process restarts.

The algorithm used by λCLAM interprets the methodical structure until it comes to the first proof method or critic. If this is applicable, then it is applied to create a new planning state, and the search continues according to the proof control. The interpretation generally uses depth-first search. The interpretation of methodicals is not very different from the interpretation of tacticals, except that it works within a system which contains additional contextual information for proof search and proof revision.

IsaPlanner's algorithm is similar to λCLAM's with the addition that the agenda of reasoning states can be manipulated by proof planning operators that allow the planner to switch between search strategies. This power has not yet been properly evaluated, but it does not appear radically different from the use of control rules to select the next appropriate goal, method or action.

4 Discussion and Future Work

Our categorisation of proof methods as operations which describe manipulations of goals *and* the context suggests there could be proof methods which act on the context alone. Consider representations that are not possible in the object language, for example diagrams. These could be represented in the con-

	Ωmega	λCLAM	IsaPlanner
Proof Planning State	goals, proof plan, control information (control rules), proof context (constraint stores), history	proof plan, control information (continuation), proof context (annotations)	proof plan, control information (continuation), proof context
Proof Language	higher order natural deduction typed λ calculus	sequent based λ calculus	various Isabelle logics
Proof Plans	DAG, verification as expansion	tree, verification in principle postponed	tree, verification interleaved
Proof Methods	declarative objects with slots: input, output, parameters, preconditions		subset of reasoning techniques
(declarativity)	slots fixed, interpreted language for preconditions	slots fixed, programming language for preconditions	slots not fixed, optional precondition-like statements in context
Proof Revision	strategic control rules decide how to react to failure	proof critics for failing methods/proof plans	
Proof Control	control rules, strategies and strategic control rules	fixed methodical language to combine methods	extensible methodical language to combine methods
(heuristic knowledge)	heuristic knowledge not in proof methods but in control rules	heuristic knowledge in proof methods, not in control rules	
Planning Algorithm	depth first search		proof methods can change local search strategy

Table 1
Ωmega, λCLAM and IsaPlanner at a glance.

text, and diagrammatic reasoning would correspond to proof methods which manipulate only the context. Such use of proof context will require further clarification of the different types of context information and the structure of context.

Our analysis also raises questions about the extent to which declarativity is desirable both in proof methods and in proof control, and the extent to which the users should be encouraged to separate legal and heuristic information.

A key issue to resolve in further investigation of these questions is the extent to which it is possible to emulate the behaviour of one style of operator with another. Our preliminary work in this direction indicates that this is possible in both direction. So methodicals and control rules are two representations of the same knowledge. Our immediate intention is to explore these questions of power and expressivity in more detail.

We have presented a framework for the discussion of proof planning in terms of Planning State, Proof Languages, Proof Plans, Proof Methods, Proof Revision, Proof Controls and Planning Algorithms. We showed that these categories are both adequate for describing all the salient features of modern proof planning systems and illuminating in that they allow like to be compared with like (even in the face of confusing and overloaded terminology).

As a case study we have applied our framework to three modern proof planning systems, IsaPlanner, Ωmega and λCLAM. This has revealed that a key area in proof planner design is the nature of proof control and context information. We hope to explore these design issues in future work.

References

[1] A. Bundy. The use of explicit plans to guide inductive proofs. E. Lusk and R. Overbeek, eds, *CADE-9*, LNCS 310, pp. 111–120. 1988.

[2] A. Bundy. The Automation of Proof by Mathematical Induction. A. Robinson and A. Voronkov, eds, *Handbook of Automated Reasoning*, Elsevier, 2001.

[3] A. Bundy. A critique of proof planning. A. C. Kakas and F. Sadri, eds, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski*, LNCS 2408, pp. 160–177. 2002.

[4] A. Cohen, S. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. In F. Baader, ed, *CADE-19*, LNAI 2741, pp. 258–273. 2003.

[5] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. F. Baader, ed, *CADE-19*, LNCS 2741, pp. 279–283. 2003.

[6] A. Ireland. The use of planning critics in mechanizing inductive Proofs. A. Voronkov, ed, *LPAR*, LNAI 624, pp. 178–189. 1992.

[7] A. Meier, M. Pollet, and V. Sorge. Comparing approaches to the exploration of the domain of residue classes. *JSC, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, 2002. S. Linton and R. Sebastiani, eds.

[8] E. Melis and J. H. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[9] Omega Group. Proof development with Ωmega. A. Voronkov, ed, *CADE-18*, LNAI 2392, pp. 144–149, 2002.

[10] L. C. Paulson. *Isabelle: A generic theorem prover.* LNCS 828. 1994.

[11] J. D. C. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with lambdaclam. C. Kirchner and H. Kirchner, eds, *CADE-15*, LNCAI 1421, pp. 129–133. 1998.