Proof Planning

Alan Bundy

January 28, 1998

Abstract

We describe *proof planning*, a technique for the global control of search in automatic theorem proving. A proof plan captures the common patterns of reasoning in a family of similar proofs and is used to guide the search for new proofs in this family. Proof plans are very similar to the plans constructed by plan formation techniques. Some differences are the non-persistence of objects in the mathematical domain, the absence of goal interaction in mathematics, the high degree of generality of proof plans, the use of a meta-logic to describe preconditions in proof planning and the use of annotations in formulae to guide search.

1 Introduction

The main research problem in automating theorem proving is the *combinatorial explosion*. A mathematical theory and conjectured theorems of it can both be represented in a computer using mathematical logic. Proofs can be constructed by applying logical rules forwards to the axioms of a theory in the hope of generating the conjecture — or, more often, by applying the rules backwards to the conjecture in the hope of reducing it to axioms. The combinatorial explosion occurs because there is choice, *i.e.* more than one rule can be applied to each of the initial and intermediate expressions. We must use *search* to be sure to try out all the possibilities. The number of intermediate expressions we must generate grows super-exponentially with the length of the desired proof. The storage and time requirements to find a proof by exhaustive search are so large that it is infeasible to construct the proofs of non-trivial theorems. Some intelligence is needed to guide the search for a proof along promising paths, avoiding less promising ones. Since human mathematicians can often conquer the combinatorial explosion and find complex proofs of hard theorems, it is promising to study human proof methods as a source of inspiration for automatic methods of proof.

At Edinburgh we have pioneered a technique for guiding the search for a proof, which we call *proof planning*, [Bundy, 1991]. A *proof plan* captures the common patterns of reasoning in families of similar proofs. It is used to provide a global control strategy for finding new proofs from the same family. Proof planning contrasts with the more local heuristics which have previously been used for search control. That is, instead of making a separate decision at each choice point, based on local clues, proof planning has some sense of the overall direction of the proof. This seems to accord more with the intuitions of human mathematicians that they first make a global plan of a proof and then fill in the details.

In this paper we survey our work on proof planning. We then go on to compare and contrast this with traditional AI work on plan formation.

2 The Nature of Proof Plans

We have analysed a large number of proofs, especially from the area of inductive reasoning. Common patterns of reasoning were identified and represented computationally as proof plans.

Our proof planning is implemented using three kinds of object:

Tactics: are computer programs which construct part of a proof by applying rules of inference in a theorem proving system, [Gordon *et al*, 1979]. A simple tactic might apply only a single rule of inference; a composite tactic will be defined in terms of simpler tactics and might construct a whole proof. Tactics are hierarchical; some tactics unpack into sub-tactics. Tactics can be composed from rules and sub-tactics sequentially, iteratively and conditionally.

Methods: are specifications of tactics. In particular, a method describes the preconditions for the use of a tactic and the effects of using it. These preconditions and effects are syntactic properties of the logical expressions manipulated by the tactic and are expressed in a meta-logic.

Critics: capture common patterns of failure of methods and suggest patches to the partial proof, [Ireland, 1992, Ireland & Bundy, 1996]. A critic is associated with a method and is similar in structure, except that its preconditions describe a situation in which the method fails. Instead of effects it has instructions on how to patch the failed proof plan.

Our *Clam* proof planner uses methods to construct a customised tactic for the current conjecture. It combines general-purpose tactics so that the effects of earlier ones achieve the preconditions of later ones. This customised tactic is then used by our proof editors, *Oyster* or *Mollusc*, to try to prove the conjecture. Sometimes the preconditions of a method succeed, but those of one its sub-methods fail. In this case a critic may suggest a patch to the proof plan. This productive use of failure via critics is made possible by proof planning and is one of its most powerful features.

Proof planning combines two previous approaches to automated reasoning: the use of tactics and the use of meta-level control. The meta-level control is used to identify appropriate tactics and to suggest patches when they fail. Proof plans abstract the proof, revealing the key steps and the structure of the proof. This abstraction can be used to construct explanations of successful proofs and give reasons for the failure of unsuccessful ones.

3 Implementation

Our implementation of proof plans consists of the following parts.

1. An object-level interactive proof editor, which can be driven by *tactics*. Initially, we built *Oyster*, [Bundy *et al*, 1990], a proof editor for constructive type theory closely modelled on Nuprl, [Constable *et al*, 1986]. More recently, we have built *Mollusc*, [Richards *et al*, 1994], a generic proof editor, *i.e.* one which takes a logic as input and becomes a proof editor for that logic.

2. A variety of logics implemented in *Mollusc*, including logics which are: first order and higher order; typed and untyped; constructive and classical.

3. A set of general-purpose tactics for controlling *Mollusc* and/or *Oyster*. A method for each of these tactics. A set of proof *critics* for each method.

4. A plan formation program, *Clam*, [Bundy *et al*, 1990], for reasoning with these methods and critics in order to build a customised tactic for each conjecture out of the general-purpose tactics.

5. A co-operative interface to *Clam*, called *Barnacle*, [Lowe *et al*, 1995]. This uses proof planning to explain the proof to the user and assist him/her in interacting with the proof process.

The *Clam* system and the proof editors have been tested on a large corpus of theorems drawn from the literature, with very encouraging results, [Bundy *et al*, 1991, Ireland & Bundy, 1996]. The planning search space is typically many orders of magnitude smaller than the object-level search space. Furthermore, the heuristics represented in the preconditions of the methods ensure that backtracking during planning is rare. So the search for a plan is computationally very cheap.

The cost of this dramatic pruning of the object-level search space is that the planning system is incomplete. Fortunately, this has not proved a serious limitation; the *Clam* system finds proof plans for a high percentage of the theorems tested and these plans are turned into proofs by *Oyster* or *Mollusc*. Thus the proof planning has proved to be a very effective way of tackling the combinatorial explosion in automatic theorem proving. Proof planning sometimes fails to find a proof. In this case the interaction provided by *Barnacle* is at a much higher level than that normally available from interactive provers. It can use the proof plan to explain the global structure of the proof and the nature of the plan failure. This can help suggest an appropriate patch.

4 Applications to Formal Methods

Formal methods of system development use mathematics to reason about computer programs or electronic circuits. This reasoning includes: verifying that a system meets a specification of its intended behaviour; synthesising a system which meets such a specification; and transforming one system into a more efficient one meeting the same specification. Use of formal methods improves the reliability, safety and security of IT systems. Unfortunately, formal methods are not as widely used as they might be due to the high skill levels and long development times required to apply them. Machine assistance is available, but even then a high level of very skilled user interaction is usually required. By automating much more of the proof obligations on formal methods using proof planning we hope significantly to reduce both the development times and skill levels required. This will make the application of formal methods more feasible and widespread.

Mathematical induction[1] is required for reasoning about objects or events containing repetition, *e.g.* computer programs with recursion or iteration, electronic circuits with feedback loops or parameterised components. Since repetition is ubiquitous and inductive reasoning is the most difficult to automate, we have focussed on the construction of proof plans for mathematical induction. Our inductive proof plans are very successful. They guide the search for quite complex proofs with a high degree of success and a very low branching rate. For instance, a recent success was to verify that the Gordon Computer (a complete microprocessor) met its specification, [Cantu *et al*, 1996].

5 Inductive Proof

Inductive proofs are characterised by the application of induction rules, of which a simple example is Peano induction:

$$\frac{P(0),\ \forall n{:}nat.\ (P(n) \rightarrow P(n+1))}{\forall n{:}nat.\ P(n)}$$

i.e. if a formula can be proved in the case $n = 0$ (the base case) and whenever it can be proved for n it can be proved for $n + 1$ (the step case) then it can be proved for all n.

Peano induction is merely the simplest and best known inductive rule of inference. Similar inductive rules are available for every kind of recursively defined data-structure, *e.g.* integers, lists, trees, sets, *etc.* For instance, the analogous rule for lists is:

$$\frac{P(nil),\ \forall h:\tau, t{:}list(\tau).\ (P(t) \rightarrow P([h|t]))}{\forall l{:}list(\tau).\ P(l)}$$

where $[h|t]$ means h consed onto t, $l : \tau$ means l is of type τ and $list(\tau)$ is the type of lists of objects of type τ.

Moreover, it is not necessary to traverse such data-structures in the obvious, stepwise manner; they can be traversed in any order, provided it is *well-founded*, *i.e.* provided there is no

[1]Not to be confused with the learning form of induction.

infinite ordered sequence of smaller and smaller elements. Nor is induction restricted just to data-structures, for instance, it is possible to induce over the control flow of a computer program or the time steps of a digital circuit.

All of these forms of induction are subsumed by a single, general schema of well-founded induction[2]:

$$\frac{\forall x{:}\tau.\ (\forall y{:}\tau.\ x \succ y \rightarrow P(y)) \rightarrow P(x)}{\forall x{:}\tau.\ P(x)}$$

where $\succ$ is some well-founded relation on the type τ, *i.e.* there is no infinite sequence: $a_1 \succ a_2 \succ a_3 \succ \ldots$. The data-structure, control flow, time step, *etc.*, over which induction is to be applied, is represented by the type τ. The inductive proof is formalised in a many-sorted or typed logical system.

Success in proving a conjecture, P, by well-founded induction is highly dependent on the choice of x, τ and $\succ$. For many types, τ, there is an infinite variety of possible well-orderings, $\succ$. Thus choosing an appropriate induction rule to prove a conjecture is one of the most challenging search problems to be solved in automating inductive inference.

6 Heuristics for Inductive Inference

The automation of inductive inference raises a number of unique difficulties in search control. These are:

Synthesis of Induction Rules: To prove a theorem by induction, one of the infinite number of possible induction rules must be synthesised.

Conjecturing Lemmata: Sometimes a lemma required to complete the proof is not already available and must be conjectured and then proved.

Generalisation of Induction Formulae: Sometimes a theorem cannot be proved without first being generalised.

In addition to these special search problems all the standard problems also manifest themselves, *e.g.* deciding if and when to make a case split, determining the witness for an existential quantifier.

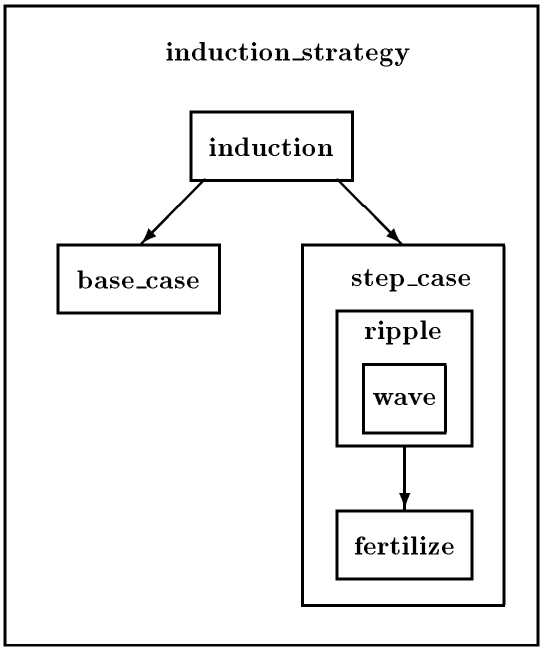
We have tried to solve these search control problems by designing tactics, methods and critics. A few such tactics can prove most of the standard inductive theorems we have tested them on. A pictorial representation of our main proof methods for inductive proof is given in figure 1.

Rippling is the key tactic in our proof plans, [Bundy *et al*, 1993, Basin & Walsh, 1996]. The places at which the induction conclusion differs from the induction hypothesis are marked by special annotations called *wave-fronts*. These can be calculated automatically by an algorithm called *difference matching*. The wave-fronts are then moved out of the way so that a copy of the induction hypothesis appears within the induction conclusion. This movement is effected by special rewrite rules called *wave-rules*. The induction hypothesis can then be applied to the induction conclusion by a tactic call *fertilize*. Rippling involves little or no search. An example is given in figure 2.

7 Comparison with Plan Formation

Work on plan formation in AI goes back to the 60s with Cordell Green's work on QA3 and Nilsson's on STRIPS. Is there any similarity between this work on plan formation and that described above?

[2] Also known as nœtherian induction.



Each of the boxes represents a method. The nesting of the boxes represents the nesting of methods, i.e. *an inner method is a sub-method of the one immediately outside it. The induction strategy is a method for a complete application of induction. After the application of induction the proof is split into one or more base and step cases (one of each is displayed here). Within the step case method, rippling is used to reduce the difference between the induction conclusion and induction hypothesis. The rippling method consists of repeated applications of the wave method. The rewritten induction conclusion is then fertilized with the induction hypothesis.*

Figure 1: A Proof Plan for Induction

7.1 Correspondence between Proof Planning and Plan formation

Plan formation builds a sequence of actions to be applied by an automated agent, *e.g.* a robot. Proof planning builds a customised tactic for guiding a proof. So some apparent differences are that proof planning does not reason about time, does not involve a robot and does not deal with the manipulation of real world objects. However, these differences are superficial. The individual mathematical rule applications in proof planning correspond to actions in plan formation. Tactics correspond to actions that expand into sub-actions during hierarchical planning. The objects being manipulated in proof planning are mathematical expressions. Proof plans are applied by an automated agent: the proof editor. The sequence of manipulations do define some notion of time, *e.g.* we could regard each successive formula in the ripple in figure 2 as being a world state at a different moment in time. So proof planning can be viewed as plan formation in the traditional AI sense. Even the concept of critic has its origins in plan formation. It was originally developed by Sussman and Sacerdoti and played a similar role to our critics in patching failed plans, [Sussman, 1975, Sacerdoti, 1977].

7.2 Persistence of Objects

One real difference is that there is no necessary persistence of objects in proof planning, whereas there usually is in plan formation. In figure 2 some parts of the expression do seem to persist. For instance, throughout the ripple there is one h, t and l on each side of each equation. The compound objects, however, come and go, *e.g.* $[h|t]$ occurs twice in the first formula, but then disappears not to reappear. In some proofs, atomic objects can behave the same way. For instance, if the rule $X + X \Rightarrow 2 \times X$ were applied then two occurrences of X would merge into one. If the rule $0 \times X \Rightarrow 0$ were applied then X would disappear altogether. The non-persistence of objects means that the mathematical world cannot be described merely by sets of relations between objects, since this assumes they persist over time. Fortunately, the mathematical formulae themselves can be used to represent the world state.

7.3 Goal Interaction

Goal interaction is a major issue in plan formation and has led to the development of non-linear planning, [Tate, 1977, Sacerdoti, 1977]. Goal interaction has not, so far, been a major issue in proof planning, so we have not needed to use non-linear planning.

Something a little like it does arise in our work on critics. For instance, the point at which a proof breaks down may not be the point at which the proof patch needs to be applied. For instance, we might detect the need for a case split, but this split might need to be made at an earlier stage in the proof than where the need for it is detected, [Ireland, 1992]. Similarly, with changes to the form of induction or generalizations of the theorem. The critics currently do this by restarting the proof from an earlier point. It is possible that a non-linear proof representation might facilitate the incremental growth, patching and reorganisation of proofs.

7.4 Differences in Domains

There are also differences in the domains in which plan formation and proof planning work. Plan formation must deal with incomplete and uncertain knowledge about the world and the automated agent may have to work with collaborators and/or against opponents. In applications to mathematics, our knowledge is certain, *i.e.* the axioms, rules *etc* have binary truth values with no degrees of certainty. Nor are there other agents assisting or opposing our proof attempts. Proof planning, however, can be adapted to deal with incompleteness, uncertainty, collaborators and opponents. We have successfully tested proof planning by applying it to non-mathematical domains, including the card game bridge, [Frank *et al*, 1992]. In the bridge domain, incompleteness arises because you do not know what cards are held by the other players, although you can sometimes make an uncertain guess based on defeasible inferences from the bidding and play. Players collaborate with one other player and are opposed by two others.

7.5 Hierarchical Planning

As illustrated in figure 1, proof plans are hierarchical. The tactics of proof planning correspond to the action expansions of plan formation. Like actions, tactics may unpack into several sub-tactics which may, in turn, unpack into sub-sub-tactics. The unpacking may involve recursion or conditionals. There do, however, seem to be some differences between the realisation of hierarchical planning in proof planning and plan formation.

Firstly, we use only a few tactics (about a dozen) compared to the much larger number typically used in plan formation. Our tactics are much more general. Rippling, for instance, is not only used in all inductive proofs, but we have also found applications in other areas of mathematics, *e.g.* for summing series, [Walsh *et al*, 1992], and for proving limit theorems in analysis, [Yoshida *et al*, 1994]. Action expansions, on the other hand, are usually more limited in their potential applications. Each ripple uses a different set of wave-rules and the number of rules used varies according to need. Newly proved conjectures may be automatically recognised as wave-rules, annotated and used in future ripples. Action expansions usually incorporate a fixed set of lower level actions. Tactics in mathematical domains can involve search and may not be guaranteed to terminate, *e.g.* they may call complete theorem provers which are semi-decision procedures. Action expansions are usually deterministic and terminating.

Secondly, and crucially, tactics have meta-logical preconditions, whereas action expansions have object-level preconditions. Our rules and axioms describe properties of numbers, lists, trees and other data-structures, *i.e.* they are object-level. The preconditions of our tactics describe syntactic properties of mathematical formulae, *i.e.* they reason *about* the object-level representation. We have developed a meta-logic for representing these syntactic properties. For instance, the definition of a wave-rule discusses the similarities and differences between the left- and right-hand sides of the rule. The preconditions of the wave tactic require wave-fronts to be present within the goal and for the matching of rule and goal to include the alignment of their wave-fronts. Such meta-level preconditions permit a level of generality for tactics which would not otherwise be possible.

Similarly, our critics have meta-level preconditions, in contrast to critics from plan formation, which have object-level critics. Whereas plan formation critics resolve conflicts within a partial plan by reordering existing actions, proof planning critics detect and patch failed proof attempts by suggesting new methods of proof.

7.6 Search Control

Search is controlled in proof planning by a combination of tactics and meta-level reasoning. Meta-level reasoning has a long history in AI and was first used in plan formation in MOLGEN, [Stefik, 1981]. Control knowledge referring to the global development of a plan has also been used in plan formation. It was first suggested for SOAR [Laird *et al*, 1987] and further explored in Prodigy [Minton *et al*, 1989]. One way in which proof planning extends this is in its use of annotation to guide search, *i.e.* the use of wave-fronts to guide rippling to preserve some parts of a formula while moving others in a desired direction.

The depth and complexity of reasoning arising in mathematics is typically greater than that required in plan formation. The attention that has been paid to search control issues in proof planning reflects this greater demand for an efficient solution.

8 Conclusion

In this paper we have described proof planning and compared it to plan formation. Proof planning combines the use of tactics with meta-level control. It is implemented by: *tactics*, which correspond to action expansions; *methods*, which specify tactics using a meta-logic; and *critics*, which capture common patterns of failure and suggest patches to the plan. Proof plans abstract the global structure of a proof and this can be used to guide search, explain the proof and suggest patches when the plan fails.

Our main application area is inductive proof, especially as used to reason about programs and hardware. Our proof plans for mathematical induction are both restrictive and successful; they require very little search in finding quite complex proofs, but have a high success rate.

The main differences between proof planning and plan formation are:

- the non-persistence of objects in the mathematical domain;
- the absence of goal interaction in the mathematical domain, and hence the adequacy of linear planning;
- the high degree of generality of proof plans;
- the use of a meta-logic for describing the preconditions of tactics, and
- the use of meta-level annotations to guide search.

It would be interesting to explore whether some of the ideas developed in proof planning can be imported into plan formation, and *vice versa*.

Acknowledgements

The research reported in this paper was supported by EPSRC grant GR/J/80702 and ARC grant 438. I am grateful for feedback on this paper from Erica Melis, Austin Tate and Andrew Ireland.

References

[Basin & Walsh, 1996] Basin, David and Walsh, Toby. (1996). A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180.

[Bundy, 1991] Bundy, Alan. (1991). A science of reasoning. In Lassez, J.-L. and Plotkin, G., (eds.), *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press. Also available from Edinburgh as DAI Research Paper 445.

[Bundy *et al*, 1990] Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al*, 1991] Bundy, Alan, van Harmelen, Frank, Hesketh, Jane and Smaill, Alan. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al*, 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567.

[Cantu *et al*, 1996] Cantu, Francisco, Bundy, Alan, Smaill, Alan and Basin, David. (1996). Experiments in automating hardware verification using inductive proof planning. Research Paper 828, Dept. of Artificial Intelligence, University of Edinburgh, Also in Proceedings of the Formal Methods for Computer-Aided Design Conference, 1996.

[Constable *et al*, 1986] Constable, R. L., Allen, S. F., Bromley, H. M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall.

[Frank *et al*, 1992] Frank, I., Basin, D. and Bundy, A. (1992). An adaptation of proof-planning to declarer play in bridge. In *Proceedings of ECAI-92*, pages 72–76, Vienna, Austria. Longer Version available from Edinburgh as DAI Research Paper No. 575.

[Gordon *et al*, 1979] Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag.

[Ireland & Bundy, 1996] Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[Ireland, 1992] Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture

Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

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

[Lowe *et al*, 1995] Lowe, H., Bundy, A. and McLean, D. (1995). The use of proof planning for co-operative theorem proving. Research Paper 745, Dept. of Artificial Intelligence, University of Edinburgh, To appear in the special issue of the Journal of Symbolic Computation on graphical user interfaces and protocols.

[Minton *et al*, 1989] Minton, S., Knoblock, C., Koukka, D., Gil, Y., Joseph, R. and Carbonell, J. (1989). Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[Richards *et al*, 1994] Richards, B. L., Kraan, I., Smaill, A. and Wiggins, G. A. (1994). Mollusc: a general proof development shell for sequent-based logics. In Bundy, Alan, (ed.), *12th Conference on Automated Deduction*, pages 826–30. Springer-Verlag. Lecture Notes in Artificial Intelligence, vol 814; Also available from Edinburgh as DAI Research paper 723.

[Sacerdoti, 1977] Sacerdoti, E. D. (1977). *A Structure for Plans and Behaviour*. Artificial Intelligence Series. North Holland, Also as SRI AI Technical note number 109, August 1975.

[Stefik, 1981] Stefik, M. (1981). Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16:141–170.

[Sussman, 1975] Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. Artificial Intelligence Series. North Holland.

[Tate, 1977] Tate, A. (1977). Generating project networks. In Reddy, R., (ed.), *Proceedings of IJCAI-77*, pages 888–893, Boston, Ma. International Joint Conference on Artificial Intelligence.

[Walsh *et al*, 1992] Walsh, T., Nunes, A. and Bundy, Alan. (1992). The use of proof plans to sum series. In Kapur, D., (ed.), *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

[Yoshida *et al*, 1994] Yoshida, Tetsuya, Bundy, Alan, Green, Ian, Walsh, Toby and Basin, David. (1994). Coloured rippling: An extension of a theorem proving heuristic. In Cohn, A. G., (ed.), *In proceedings of ECAI-94*, pages 85–89. John Wiley.

Conjectured Theorem:

$$\forall K, L{:}list(\tau).\ rev(K <> L) = rev(L) <> rev(K)$$

Induction Hypothesis:

$$rev(t <> l) = rev(l) <> rev(t)$$

Induction Conclusion and Ripple :

$$\begin{aligned}
rev(\,\boxed{[h|t]}^{\uparrow} <> l) &= rev(l) <> rev(\,\boxed{[h|t]}^{\uparrow}\,) \\
rev(\,\boxed{[h|t <> l]}^{\uparrow}\,) &= rev(l) <> \boxed{rev(t) <> [h]}^{\uparrow} \\
\boxed{rev(t <> l) <> [h]}^{\uparrow} &= \boxed{rev(l) <> rev(t) <> [h]}^{\uparrow} \\
&\boxed{rev(t <> l) = rev(l) <> rev(t) \wedge [h] = [h]}^{\uparrow}
\end{aligned}$$

The conjecture to be proved is that list reversal, rev, distributes over list append, $<>$, *with a switch in the argument order. We use the Prolog convention that variables are in upper case and constants in lower case. We also use the Prolog notation for list cons,* i.e. $[Hd|Tl]$. $X{:}list(\tau)$ *means that* X *is a list of objects of type* τ.

The proof is by induction on K. *Only the step case is shown. The induction conclusion is annotated to show its similarities to and differences from the induction hypothesis. The differences are the bits which are inside a shaded area. These are called* wave-fronts. *The unshaded bits inside the shaded boxes are called* wave-holes. *Rippling makes the wave-fronts bigger and bigger until a copy of the induction hypothesis appears wholly within a wave-hole. When this happens this part of the induction conclusion can be replaced with true and the remaining goal is* $[h] = [h]$, *which is trivially true. We call this final step* fertilization.

Rippling is achieved by applying wave-rules. *The wave-rules used in this example can be found in figure 3. Rules (1) and (2) are applied to the left- and right-hand sides, respectively, of the first line, then rules (2) and (3) to the second line, and finally rule (4).*

Figure 2: An example of rippling

$$\boxed{[Hd|Tl] <> L}^{\uparrow} \Rightarrow \boxed{[Hd|Tl <> L]}^{\uparrow} \tag{1}$$

$$rev(\boxed{[Hd|Tl]}^{\uparrow}) \Rightarrow \boxed{rev(Tl) <> [Hd]}^{\uparrow} \tag{2}$$

$$X <> \boxed{Y <> Z}^{\uparrow} \Rightarrow \boxed{(X <> Y) <> Z}^{\uparrow} \tag{3}$$

$$\begin{aligned} \boxed{K_1 <> K_2}^{\uparrow} &= \boxed{L_1 <> L_2}^{\uparrow} \\ &\Rightarrow \boxed{K_1 = L_1 \wedge K_2 = L_2}^{\uparrow} \end{aligned} \tag{4}$$

Wave-rules are also annotated to show the similarities and differences between the left- and right-hand sides of the rules. To apply a wave-rule, the left-hand side is matched against a subexpression of the induction conclusion, which is then replaced with the right-hand side. This matching includes alignment of the annotations as well as the normal matching. This additional condition severely restricts the choices and reduces search. A measure is defined using the height of the wave-fronts. By definition this measure must be strictly less on the right-hand side of each rule than on the left-hand side.

Rules (1) and (2) come from the recursive definitions of $<>$ *and* rev*, respectively. Rule (3) is the associativity of* $<>$*. Rule (4) is the equality congruence rule for* $<>$*. Note that rule (4) is an implication from right to left, but because Clam reasons backwards, the direction of rewriting is inverted. In the example of figure 2, a weakened form of rule (4) is used, in which a wave-hole on the right-hand side of each equality is shaded and becomes part of the wave-front. When proved, the conjecture of figure 2 will also form a wave-rule.*

Figure 3: Examples of wave-rules