How to Solve it by Induction *

Alan Bundy

Department of Artificial Intelligence
University of Edinburgh

Abstract

We describe a computer program that proves theorems using mathematical induction. It is used to prove properties of computer programs. Helpful tips guide the program's search for a proof. These tips are used to form *proof plans,* which describe the structure of proofs at multiple levels.

We are attempting to discover general-purpose proof plans which describe the common structure in families of similar proofs. This study constitutes a new *science of reasoning* in which mathematical proofs are the objects of analysis.

1 Introduction

Generations of mathematics students have found inspiration in George Polya's book "How to Solve It", [Pólya 45]. This marvellous little book is packed with useful tips for solving mathematical problems. Polya invites the problem solver to ask a series of questions designed to stimulate thinking, for instance "What is the unknown?", "What are the data?", "Do you know a related problem?".

In this paper we describe a computer program which solves mathematical problems. We will see that, just like human mathematicians, our artificial mathematician needs helpful tips to guide its problem solving. Unfortunately, Polya's tips are too vague and general to be used directly in our computer program. We have had to devise much more precise and specific tips.

The *Oyster/Clam* system is a computer program for proving theorems by mathematical induction, [Bundy *et al.* 91b]. *Oyster* is the part of the system responsible for making sure the proof steps are legal; *Clam* is the part of the system responsible for guiding *Oyster* by suggesting which legal steps to make. *Clam* embodies a collection of helpful tips that we have devised for guiding inductive proofs.

We have studied a large number of simple inductive proofs and observed that they all have a similar overall structure. We call it an inductive *proof plan.* This proof plan consists of a series of tips on what to do next and how to do it, at each stage in the proof. I now use it myself when proving inductive theorems 'by hand'. Other mathematicians might also find it useful. Our artificial mathematician uses it. It is represented in the computer and is used by *Clam* to guide *Oyster*.

2 Formal Methods in Computer Program Design

Simple inductive proofs play a key role in the use of formal methods in the design of computer programs. Using formal methods a program can be proved to satisfy a

*The research reported in this paper was supported by SERC grant GR/F/71799, and an SERC Senior Fellowship to the author. I would like to thank David Basin, Ina Kraan, Pete Madden and Toby Walsh for discussions about this paper.

specification of the behaviour required of it; or a program can be synthesised to meet a specification; or a program can be transformed into an equivalent but more efficient program. Each of these tasks generates proof obligations. For instance, suppose I want to write a computer program for sorting a list of names into alphabetical order. The specification of this program might be that the output list contains precisely the same elements as the input list, but in the correct order (*i.e.* the output is an ordered permutation of the input). To verify a particular sort program I would prove mathematically that its output was an ordered permutation of its input. Most computer programs include repetition, either in the data structures or the control structures. Proofs about programs with repetition usually require mathematical induction.

Formal methods like these are needed to bring reliability into program design. Designers of bridges and buildings have used mathematical techniques for many years. As a result bridges and buildings are a lot less likely to fall down than they used to be. Unfortunately, computer program design is still a black art. As a result, computer programs are notoriously unreliable. Computers are increasingly being used in situations in which their breakdown can cause injury or loss of life or can jeopardise security. Unreliability is no longer tolerable. Formal methods offers one solution.

A major obstacle to the widespread adoption of formal methods is the skill required to prove the theorems that arise. The most promising solution to this problem is to provide machine assistance to the programmer. The greater the machine's ability to prove theorems the more assistance it can provide. Unfortunately, today's mechanical assistants are only able to prove small parts of the proof, leaving the hardest parts to the human user, and thus restricting the use of formal methods to a small coterie of mathematically sophisticated programmers.

We are testing the *Oyster*/*Clam* system on the conjectures that arise from formal program design. Proof plans promise to automate much more of the proof process than was hitherto possible. We hope this will enable the more widespread use of formal methods and improve the reliability of computer programs.

3 Functional Programs

In order to use the tools of mathematics to prove theorems about computer programs we must have some way of turning programming problems into mathematical problems. Many ways have been proposed of doing this for many different programming languages. The most modern and the simplest to understand is to view a computer program as a mathematical theory. In logic programming a computer program consists of a set of logical formulae which are the axioms of a logical theory. In functional programming a computer program is a set of equations which are the axioms of an algebra. This viewpoint is something of an oversimplification. Some aspects of programming are not so easily represented as mathematical expressions, *e.g.* input/output. For the purposes of this paper we will confine our attention to those aspects that can.

Consider, for instance, the following definition of addition on the natural numbers. It consists of a set of two equations:

$$0 + m = m \tag{1}$$
$$s(m) + n = s(m + n) \tag{2}$$

The natural numbers are represented here in a unary notation as: 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, ..., *e.g.* $s(s(0))$ represents 2 and $s(s(s(s(0))))$ represents 4. This is not a very practical data-structure for representing numbers, but it will serve as a simple illustration of the ideas. A more practical representation would introduce distracting complications without changing the essential story.

Equations (1) and (2) define a functional computer program for performing addition. The equations are used as *rewrite rules*. Whenever all or part of an expression to be evaluated matches the left-hand-side of one of these two rules then it is rewritten to match the right-hand-side. For instance, suppose we wanted to add 2 and 2. This problem would be represented as the expression $s(s(0)) + s(s(0))$ and evaluated as follows:

$$\begin{array}{rcll} s(s(0)) + s(s(0)) & = & s(s(0) + s(s(0))) & \text{by equation (2)} \\ & = & s(s(0 + s(s(0)))) & \text{by equation (2)} \\ & = & s(s(s(s(0)))) & \text{by equation (1)} \end{array}$$

which gives the representation of 4, as one would hope.

Equations (1) and (2) define a *recursive* program, that is they define $+$ in terms of itself. Notice that equation (2) contains a $+$ on the right-hand-side as well as the left. Equations, like (2), in which the function being defined occurs on each side, are called the *step* equations of a recursive definition. The other equations, *e.g.* equation (1), are called the *base* equations.

There is always a danger that recursive programs will loop forever. For instance, applying the equation:

$$s(m) + n \quad = \quad s(m) + n$$

would just replace one expression with an identical one, indefinitely. Equation (2) does not cause looping. This is because the $+$ on the right-hand-side is applied to x, a smaller data-structure than $s(x)$, which $+$ is applied to on the left-hand-side. So successive rewritings using (2) will eventually apply $+$ to 0, the smallest data-structure, which (2) cannot rewrite.

4 Proving Program Properties

To prove properties of logic or functional programs we prove theorems in the corresponding logical or algebraic theories. For theorems about recursive programs we usually need to use mathematical induction. There is a duality between recursion and induction; to each kind of recursion there is a corresponding rule of induction. For instance, the induction rule that corresponds to equations (1) and (2) is:

$$\frac{P(0), \qquad P(n) \vdash P(s(n))}{P(n)} \tag{3}$$

This states that if we can prove a property P for 0 and if, assuming it for n, we can prove it for $s(n)$, then we have proved it for all n. The formula $P(n)$ which appears to the left of the $\vdash$ sign is called the *induction hypothesis* and the formula $P(s(n))$ which appears to the right is the *induction conclusion*. The variable n is called the *induction variable*.

There are many other induction rules. Two of these are shown in figure 1.

We can use induction rules, the equations defining programs, and some other simple axioms for $=$ and the logical symbols, to prove properties of programs. For instance, we might want to prove that whether three numbers are added starting from the left or from the right, we get the same result. This theorem is called the *associativity of* $+$. It can be formalised as:

$$x + (y + z) \quad = \quad (x + y) + z$$

A proof of this theorem is given in figure 2

$$\frac{P(0), \qquad P(s(0)), \qquad P(n) \vdash P(s(< s(n)))}{P(n)}$$

$$\frac{P(0), \qquad P(s(0)), \qquad prime(p) \vdash P(p), \qquad P(m) \,\&\, P(n) \vdash P(m \times n)}{P(n)}$$

In the first rule the step case goes up in steps of 2. This requires two base cases. The second rule is based on the multiplicative structure of the natural numbers. Here the base cases deal with the numbers 0, 1 and any prime number. The step case assumes the property for any two numbers and proves it for their product. The second induction rule relies on the fact that any natural number greater than 2 is either a prime or is a product of two smaller numbers.

Figure 1: More Induction Rules

5 Automated Theorem Proving

The aim of our research is to automate the discovery of proofs like that in figure 2.

The first attempts to get computers to prove mathematical theorems were made in the 1960s. They all took advantage of ideas developed in mathematical logic in the 1930s. In particular, most attempts were based on a theorem of Jacques Herbrand, which suggested an exhaustive procedure for generating a logical proof for a conjecture.

These early attempts were disappointing. In theory, if the conjecture were provable then Herbrand's procedure and its variants would eventually generate a proof. If the conjecture were not provable then the procedure might stop, but might go on forever — you could not tell which in advance. In practice, automatic theorem provers were able to prove very simple theorems, but on anything non-trivial they either exhausted the available computer memory or the patience of their users or both. The performance of automatic theorem provers was improved only marginally by improvements in computer hardware. To make a difference it was necessary to employ helpful tips to guide the provers.

To see how this is possible it is necessary to be more concrete about how automatic theorem provers work. Most provers work from the conjecture to be proved towards the axioms of the theory. A rule is applied backwards to this conjecture to reduce it to a (hopefully) simpler problem. Several alternative rules might be tried. Then further rules are applied to the resulting problems to produce yet simpler problems. This is repeated until one of the problems to be solved is an axiom. This process can be envisaged as growing a tree. This is illustrated in figure 3. Trees can also be used to represent proofs (where the branching represents case splits) and mathematical expressions (where the branching represents different function arguments). These three different kinds of tree should not be confused.

The problem is that the size of these search trees grows exponentially (or worse) with the length of the proof. Logical proofs consist of lots of little proof steps and tend to be quite long. To prove non-trivial theorems it is necessary to grow a search tree of astronomical proportions. Such trees exhaust the storage capacity of even the biggest computers and take an unacceptably long time to grow on even the fastest computers. This phenomenon is called the *combinatorial explosion.*

The combinatorial explosion is made worse in inductive proofs because some of

Base Case

$$
\begin{aligned}
0 + (y + z) &= (0 + y) + z \\
y + z &= y + z
\end{aligned}
$$

Step Case

$$
\begin{aligned}
x + (y + z) = (x + y) + z &\vdash s(x) + (y + z) = (s(x) + y) + z \\
x + (y + z) = (x + y) + z &\vdash s(x + (y + z)) = s((x + y) + z) \\
x + (y + z) = (x + y) + z &\vdash x + (y + z) = (x + y) + z
\end{aligned}
$$

The proof works backwards from the statement of the theorem. Induction rule (3), with x as the induction variable, is applied to give a base and step case. The base case is easily proved using equation (1) twice as a rewrite rule. This reduces it to an instance of the reflexivity axiom for $=$. The step case requires three applications of the equation (2) followed by one of the substitution axiom for s. The induction hypothesis and the induction conclusion are now identical. The step case is an instance of a simple logical axiom. This completes the step case and, hence, the whole proof.

Figure 2: The Proof of the Associativity of $+$

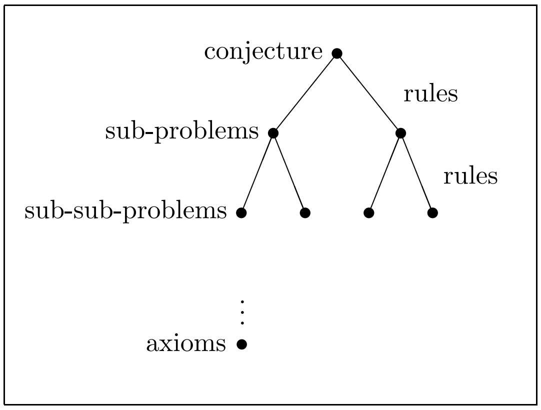
the nodes in the search tree have an infinite number of daughters. For instance, there are an infinite number of different induction rules any of which can be applied. Moreover, it is sometimes necessary to generalise the conjecture to be proved and there are an infinite number of different ways to generalise it. This means that the combinatorial explosion is encountered even when trying to prove theorems with short proofs.

The usual solution to the combinatorial explosion is to guide the search for a proof by growing only that part of the search tree that seems most likely to lead to a proof. This is where the helpful tips come in. Helpful tips in automatic theorem proving usually take one of the following forms:

- The sub-tree below a particular node can be *pruned*, *i.e.* not developed further, because it looks unlikely to lead to a proof.
- Some particular node should be developed next, because it looks most promising.
- Some particular rule should be applied next, because it looks most promising.

One characteristic of all these tips is that they are *local*, *i.e.* given the current state of the search tree, they say what to do as the next step. They do not take a global view of the proof attempt. This contrasts with the experience of human mathematicians, who often have an overall strategy in mind when searching for a proof.

Some automatic theorem provers do take a broader view of the proof attempt by embodying these helpful tips in computer programs called *tactics*. A tactic analyses



Trees in computer science are usually Australian, i.e. *they grow downwards from the root. Each node of the tree represents a problem; the lines between them represent rule applications. At the root of the tree is the conjecture to be proved. The lines descending from it are the alternative rules that can be applied to it. Each of these leads to a daughter node representing a new sub-problem. At the leaves of the tree are the axioms. Each path from the root to a leaf is a proof of the conjecture.*

Figure 3: A Search Tree

the current problem and specifies a combination of rules of inference to apply, *e.g. apply this rule then that one; if such-and-such then apply this rule, otherwise apply this one; apply this rule repeatedly until this is true;* etc. However, tactics are typically used to guide only a small part of a proof, *e.g.* the simplification of a formula or the composition of a few rules.

Our work on proof plans *does* take a more global view. It uses 'bigger' tactics, some of which can generate a complete proof. When several tactics are required these are put together in advance by analysing the conjecture to be proved.

6 Rippling Wave-Fronts

The proof of the associativity of + in figure 2 seems simple and direct and appears to require no search. However, an unguided, exhaustive automatic theorem prover would have to grow a large search tree. At every step there are several alternative rules it could have applied. In particular, it could apply one of an infinite variety of induction rules or make any one of an infinite variety of generalisations. Fortunately, fairly simple hints will prevent an automatic theorem prover being distracted from the main proof by any of these useless steps. Unfortunately, this ceases to be true if we try to get automatic proofs of more complex theorems or when we have previously proved theorems available as rewrite rules. Then we need more sophisticated hints.

The associativity proof, while very simple, does serve to illustrate some general points about the structure of inductive proofs.

- An inductive rule is applied generating some base and step cases.
- Base equations play a major role in proving base cases.
- Step equations play a major role in proving step cases.

- The aim of the step case is to use the induction hypothesis to prove the induction conclusion. This is usually done by rewriting the induction conclusion so it contains a sub-expression which matches the induction hypothesis.

The key to our inductive proof plan revolves around this last observation. We want to guide the application of rewrite rules in the step cases so that a copy of the induction hypothesis appears within the induction conclusion. We do this by marking those bits of the induction conclusion which do not appear in the induction hypothesis and then directing the rewriting so as to move them out of the way. The marked bits are called *wave-fronts* and the moving process is called *rippling*. The analogy is as follows.

> Imagine you are in Scotland standing beside a loch (see figure 4). The surrounding mountains are reflected in the loch. You throw something in the loch. The waves it makes disturb the reflection. The wave-fronts ripple outwards leaving the reflection intact again. The mountains are the induction hypothesis, the reflection is the induction conclusion and the wave-fronts are the expressions introduced into the induction conclusion by the induction rule.

In our associativity of + example the wave-fronts are the $s(\ldots)$ expressions. In general, wave-fronts are expressions with holes in them. We illustrate this by drawing boxes around the wave fronts and underlining the holes inside them. Small arrows indicate the direction in which the wave-fronts are to be rippled. Figure 5 illustrates the process of rippling on the step case of the associativity of + example. An alternative, graphical, representation of rippling is given in figure 6.

The wave-fronts are originally inserted into the induction conclusion by the application of induction. Wave-fronts are marked in the induction rule, *e.g.*

$$\frac{P(0), \qquad P(n) \vdash P(\boxed{s(\underline{n})}^{\uparrow})}{P(n)}$$

so that they are inherited into the induction conclusion when the induction rule is applied.

To ensure that the rewriting process moves the wave-fronts in the required direction and leaves the rest of the induction conclusion untouched, we restrict rewriting to a particular kind of rewrite rules that we call *wave-rules*. A wave-rule is also annotated with wave-front markers. Some example wave-rules are shown in figure 7.

When a wave-rule is applied the wave-fronts markers in the rule and the induction conclusion must be aligned. This ensures that wave-fronts are rippled outwards. Because of the sense of direction imposed by rippling there is no danger of looping, even when an equation is turned into two wave-rules: one in each direction.

The account above only serves to dip our toes into rippling. There are a number of variations on the basic idea which extend it to more complex problems. A complete account is given in [Bundy *et al.* 91a].

7 Proofs Plan for Inductive Proofs

Rippling is represented in our *Oyster/Clam* system as a tactic. It is combined with other tactics into a larger tactic for controlling the use of induction. We call this larger tactic the *induction strategy*. The other tactics that go to make up the induction strategy handle: the application of the induction rule; the use of the

Figure 4: Two Examples of Real Rippling

$$\begin{aligned}
\boxed{s(\underline{x})}^{\uparrow} + (y+z) &= (\boxed{s(\underline{x})}^{\uparrow} + y) + z \\
\boxed{s(\underline{x})}^{\uparrow} + (y+z) &= (\boxed{s(\underline{x})}^{\uparrow} + y) + z \\
\boxed{s(\underline{x+(y+z)})}^{\uparrow} &= (\boxed{s(\underline{x+y})}^{\uparrow}) + z \\
\boxed{s(\underline{x+(y+z)})}^{\uparrow} &= \boxed{s(\underline{(x+y)+z})}^{\uparrow} \\
x+(y+z) &= (x+y)+z
\end{aligned}$$

The first equation is the induction conclusion annotated by wave-fronts. Subsequent equations show the effect of rippling these wave-fronts outwards using equation (2) (three times) and finally the substitutivity of s. The wave-fronts ripple outwards until they finally disappear altogether. At this point the induction hypothesis can be used to complete the proof. In general, the wave-fronts do not disappear but are moved to the outside of the induction conclusion leaving an expression inside that matches the induction hypothesis.

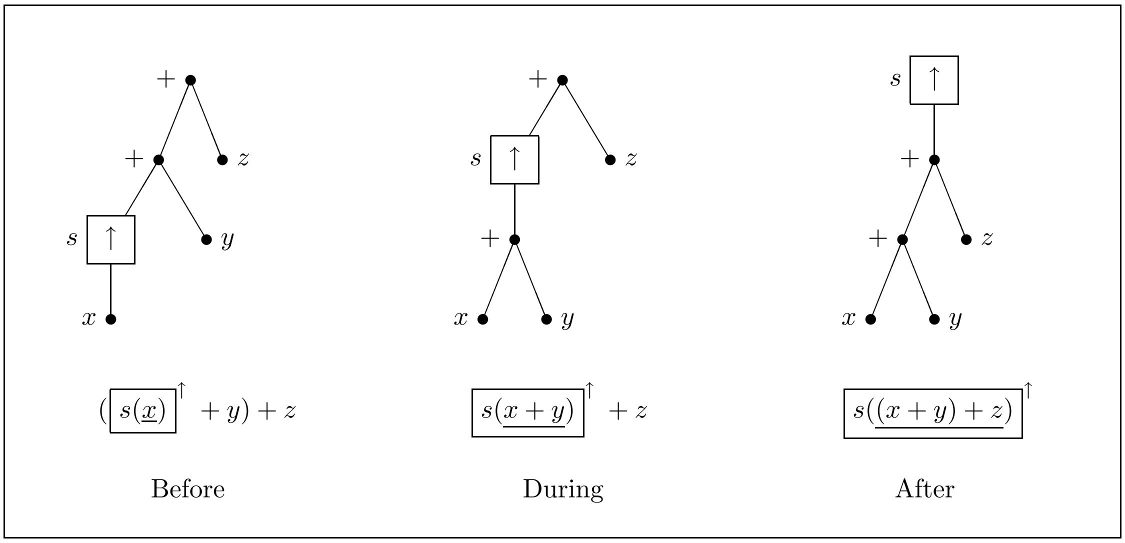
Figure 5: Rippling in the Associativity of +

induction hypothesis to prove the rippled induction conclusion; and the proof of the base case.

With each tactic we store a description of its preconditions and effects. We call this combination a *general-purpose proof plan*. The preconditions describe the kind of problem to which the tactic can be applied and the effects describe the result of applying it. These enable the *Clam* system to reason about the tactics and combine them together into a large tactic, custom-built for the current conjecture. Typically, this custom-built tactic might consist of several nested applications of the induction strategy interleaved with other tactics for handling non-inductive parts of the proof. This large tactic and the preconditions and effects which tie it together we call a *special-purpose proof plan*. Once a complete proof plan has been formed by *Clam*, the tactic it suggests is extracted and used to guide *Oyster* in constructing a proof.

The reasoning of *Clam* revolves around rippling. For instance, its choice of induction rule and induction variable is determined by whether they will permit rippling to work, *i.e.* whether there are wave-rules available to ripple the wave-fronts inserted by the induction rule.

Clam may not be able to find a proof plan to prove a conjecture, even when it is provable. For instance, rippling may be blocked by the absence of a suitable wave-rule. There are several solutions to this. One is to conjecture and prove a suitable wave-rule. Analysis of the blocked ripple can suggest a lot about the shape of the missing wave-rule. Another solution is to generalise the theorem. Again, an analysis of the blocked ripple can suggest a lot about the form the generalisation should take. We are currently investigating these and other ways to patch failed proof plans and have some partial solutions.



The three trees show the term before, during and after rippling. Each tree represents the term as an expression tree. The nodes of each tree are labelled by a function or predicate symbol or a constant or variable. A function or predicate of arity n has n descendent subtrees: one for each of its arguments. Wave-fronts are indicated by square nodes and all other nodes by dots. Note that the three trees differ from each other only in the position of the square nodes. These square nodes are higher in each successive tree; the 'After' one being beached at the top. The ↑s in the square nodes indicate the direction of rippling-out. This direction is upwards in this graphical representation of expressions or outwards in the standard one.

Figure 6: Rippling-Out $(s(x)+y)+z$

8 Results

We have tested the *Oyster/Clam* system on about 100 simple inductive theorems. In each case *Clam* was able to find a proof plan with little or no search, *i.e.* it rarely made a mistake and had to re-plan. The tactic of this plan was extracted and used to drive *Oyster* to generate a proof. This too did not require search. Without such guidance *Oyster* could only prove a couple of the very simplest theorems. This is because the search trees were so huge that our computer ran out of space. We estimated that some theorems had search trees of 10^{32} nodes in the *Oyster* logic, even without the infinite branching of induction and generalisation.

A typical example of the sort of theorem proved by *Oyster/Clam* is:

$$count(a, sort(l)) \quad = \quad count(a, l)$$

The function $count(a,l)$ counts the number of occurrences of a in a list l. The function *sort* sorts the elements in a list into order. So the theorem asserts that the number of occurrences of each element in a list is unchanged by sorting it. The proof found by *Clam* uses three inductions and seven case splits.

Rippling was built exclusively as a tactic for guiding the step cases of inductive proofs. However, we were recently surprised and delighted to discover that it could be used to guide other kinds of proofs. We have now made extensive use of rippling

$$\boxed{s(\underline{m})}^{\uparrow} + n \quad \Rightarrow \quad \boxed{s(\underline{m+n})}^{\uparrow} \tag{4}$$

$$\boxed{s(\underline{m})}^{\uparrow} = \boxed{s(\underline{n})}^{\uparrow} \quad \Rightarrow \quad m = n \tag{5}$$

$$l + (\boxed{\underline{m} + n}^{\uparrow}) \quad \Rightarrow \quad \boxed{(\underline{l+m}) + n}^{\uparrow} \tag{6}$$

$$(\boxed{l + \underline{m}}^{\uparrow}) + n \quad \Rightarrow \quad \boxed{l + (\underline{m+n})}^{\uparrow} \tag{7}$$

The $\Rightarrow$s between the left and right hand sides of the wave-rules show the direction of rewriting. The left and right-hand-sides of each wave-rule are identical except for the wave-fronts. The wave-fronts are either further out on the right-hand-side than they are on the left or they are absent altogether on the right.

*Good sources of wave-rules are: the step equations of recursive definitions (*e.g. *(4) based on equation (2)); substitution laws (*e.g. *(5)); associative laws (*e.g. *(6) and (7)) and distributive laws.*

Wave-rules (4) and (5) were used in the associativity of + proof (see figure 2). Note that rule (5) is an implication from right to left, but is applied backwards as a rewrite rule in our backwards style of proof. Wave-rules (6) and (7) show that the associative law of +, once it has been proved as a theorem, can be used as a wave-rule, in either direction.

Figure 7: Example Wave-Rules

for finding closed form solutions to series, *e.g.* given $\sum \frac{1}{i.(i+1)}$ find the equivalent expression $\frac{n}{s(n)}$ not containing the $\sum$ symbol. More details of this work can be found in [Walsh *et al.* 91]. More generally, it looks like rippling may be useful in any reasoning task where one expression must be manipulated to match another.

9 Conclusion

The global guidance provided by proof plans can avoid huge combinatorial explosions. We can represent these proof plans in a computational form so that an automatic theorem prover can use them to find proofs. In the *Oyster/Clam* system we have done this for simple inductive proof plans.

An improved ability to prove simple inductive theorems will make the use of formal methods in the design of computer programs much more feasible. Mechanical assistants for program designers will be able to automate far more of the proof obligations that arise in verifying, synthesising and transforming programs. This will reduce the level of mathematical sophistication required to use formal methods techniques, opening them up to a wider range of programmers. As a result, computer programs could become more reliable.

Our work on proof plans treats mathematical proofs as objects of scientific enquiry. We analyse proofs, trying to understand not why a step was legal but why it leads to a proof where other legal steps do not. We try to discover and describe the common structure across a family of proofs. We describe proofs at several levels simultaneously, ranging from the detailed level of legal, logical steps to the global level of the overall proof plan. This is leading us to a better understanding of what it takes to be a mathematician — both human and artificial. It could also help us in the teaching of mathematics, *e.g.* by describing to students the helpful tips used by *Clam*.

References

[Bundy *et al.* 91a] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the Journal of Artificial Intelligence.

[Bundy *et al.* 91b] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Pólya 45] G. Pólya. *How to Solve It.* Princeton University Press, 1945.

[Walsh *et al.* 91] T. Walsh, A. Nunes, and Alan Bundy. The use of proof plans to sum series. Research Paper 563, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the proceedings of CADE-11.