Can theorem proving keep the planes flying

Can Theorem Proving Keep the Planes Flying? *

Alan Bundy

January 19, 2012

Abstract

Alan Bundy is Professor of Automated Reasoning in the Department of Artificial Intelligence at the University of Edinburgh. His Mathematical Reasoning Group is applying mathematics to the development of computer programs. The synthesis, verification and transformation of computer software is done by proving mathematical theorems using an automated theorem prover. A major technical problem to be overcome is how to guide the search for a proof so the theorem prover does not become bogged down in the possibilities. To solve this problem, the Edinburgh group have developed *proof planning*. Their automated theorem prover first constructs an outline of the desired proof and then fills in the details of this outline.

1 Why Formal Methods are Needed

Computer programs are playing an increasingly important role in all our lives. More and more we are coming to rely on them. When they go wrong the consequences can vary from minor inconvenience to major catastrophe. 'Safety critical' applications of computers range from the control of complex systems like air traffic (see figure 1) and nuclear power plants to the small embedded computers in medical instruments. It is impossible to test these programs against the potentially infinite number of situations they may encounter in use. But we cannot afford for them to fail. We *must* develop a design methodology for programs which gives a high degree of assurance that they will behave correctly.

As they mature, engineering disciplines develop a range of techniques for ensuring the high quality of their products. Mathematics usually plays a major role in these techniques. For instance, consider the use of statics in predicting the stresses and strains in the components of a bridge. As it matures, Computer Science is taking a similar path. The aim is to provide a quality assurance for computer programs and to minimise the risk of spectacular or dangerous failure.

*The research reported in this paper was supported by SERC grant GR/H/23610. and an SERC Senior Fellowship to the author. I would like to thank Ian Green, Andrew Ireland, Helen Lowe, Julian Richards and Toby Walsh for feedback on any earlier draft.

Figure 1: A modern air traffic control system relies on a complex computer program. If this program failed then aeroplanes could collide. Increasingly these programs are being specified and designed with the aid of formal methods. Programs are mathematically proved to meet their specifications.

One family of techniques being developed for — and increasingly used in — program development is called *formal methods.* This is the use of mathematics to prove that programs have certain properties. A program might be proved to terminate; two programs might be proved to be equivalent; an inefficient program might be transformed into an equivalent, but more efficient, program; a program might be verified to obey some specification; or a program might be synthesised that obeys some specification. Similar techniques are also being used in the development of electronic circuits.

One barrier to the more widespread use of formal methods is the lack of appropriate mathematical skills among computer programmers and hardware designers. The Edinburgh group is addressing this problem by developing computer aids to assist formal methods users. Automated theorem proving is used to lift some of the burden of proof from the shoulders of the software/hardware developer. Since current automated theorem proving technology is not good enough, the group is improving it. In particular, they are looking at ways of automatically guiding the search for a proof. They are studying the kind of proofs that arise in formal methods to spot common patterns in families of similar proofs. These patterns, or *proof plans*, are then used to guide future proof attempts.

2 Viewing Program Development as Mathematics

In order to use the tools of mathematics to prove properties of computer programs we must have some way of turning programming problems into mathematical problems. Many ways have been proposed of doing this. The most modern and the simplest to understand is to view a computer program *as* a mathematical theory. The lines of program code are interpreted as the axioms of a new branch of mathematics in which the objects of study are computer data structures. In functional programming the programs are functions which map data structures to data structures. In logic programming the programs are relations between data structures. Proving properties of these programs consists of proving theorems in this mathematical theory.

Most types of data structure can take an infinite variety of possible forms. A finite definition of such data structures requires *recursion*, *i.e.* self reference. The following recursive definition of a list of elements gives an illustratation.

> **Base Case:** The empty list is a list.
> **Step Case:** A list with a new element added is a list.

The circularity in this definition is not vicious. Given a finite list we can always show it is a list in a finite number of steps. We repeatedly apply the step case of the definition and finish by applying the base case.

The specification of a computer program can also be viewed as a mathematical formula. Consider a program for sorting a list of names into alphabetical order. Its specification might be:

> The output list is ordered.
> The output list is a permutation of the input list.

"Ordered" and "permutation" can both be defined as mathematical relations. The question of whether a particular computer program meets this specification can then be represented as a theorem to be proved in a mathematical theory of lists.

The proofs of theorems about recursive data structures usually calls for mathematical induction. Each data type has its own forms of induction. For instance, one of the induction rules for lists is:

> To prove a property for all lists:
> **Base Case:** prove it for the empty list;
> **Step Case:** assuming it true for an arbitrary list, prove it for that list with a new element added.

This induction rule is modelled directly on the recursive definition of lists.

3 The Need for Computer Aids

Proofs of properties of programs are usually mathematically unsophisticated, but can be long and are usually in non-standard mathematical theories. This poses two problems inhibiting the more widespread use of formal methods. Firstly, most programmers do not have the mathematical background required to formulate and prove the necessary theorems. Secondly, it is easy to make errors during what can be a long and tedious process. One solution to these problems is to provide computer aids for the program developer.

The Edinburgh group have investigated two kinds of such computer aid: systems for helping programmers formulate their needs in mathematical terms and systems for helping programmers prove the resulting theorems. This article describes only the latter kind of system: automated theorem provers.

Automated theorem proving has its roots in mathematical logic. In logic a mathematical theory can be represented as a collection of axioms and of rules for deriving new theorems from axioms and from old theorems. The formulae can be represented as computer data structures and the rules can be represented as programs which manipulate formulae. Most automated theorem provers apply the rules backwards, reducing the goal to be proved into simpler sub-goals and finally into axioms.

The main problem is search. At every step many rules are applicable. Each rule produces an alternative collection of sub-goals. The different permutations define a large *search space* of alternative proof attempts. Only a very small minority of these attempts will be successful. Some of the unsuccessful branches of this search space may be infinite. A theorem proving program that searches blindly among these alternatives rapidly becomes bogged down. It runs out of storage space and it runs out of time. This phenomenon is called the *combinatorial explosion.*

The solution to the combinatorial explosion is to use *heuristics* — rules of thumb — to guide the search for a proof along the most promising paths. Most such heuristics are local and shallow. For instance, a crude measure of formula complexity might be used to identify the simplest sub-goal so that it can be developed next. In contrast, the Edinburgh group has investigated more global heuristics which rely on a deeper analysis of what is going on in the proof.

4 What are Proof Plans?

By analysing a large number of proofs of program properties, the Edinburgh group have discovered common patterns. In particular, they have noticed that most inductive proofs have a common structure. They have captured this common structure in a computational form and used it to guide the search for a proof. As a result, their automated theorem prover seldom searches. Usually, it goes straight to the proof.

Of course, there is a price to be paid for this success; the Edinburgh automated prover cannot find proofs which differ from its stored proof plans. Fortunately, in formal methods proofs, such failures are usually due to only minor divergences from a known proof plan. It is usually possible to give an analysis of the failure which can be used to patch the failed proof attempt and put the prover back on track. Ways of automating this analysis are currently being investigated, so it might be used either interactively by a human user or automatically by a proof patching system. The high-level language in which proof plans are described and their failures analysed provides a more comprehensible basis for interaction with a user than the low-level language of logic.

The flavour of the inductive proof plan can be given by describing a small but central part of it, called *rippling.* Suppose a goal is to be proved from a structurally similar hypothesis. This is the situation in the step case of an inductive proof:

> **Induction Hypothesis:** Assume the property true for an arbitrary list.
>
> **Induction Conclusion:** Prove the property for that list with a new element added.

The induction conclusion is the goal which is be proved from the, structurally similar, induction hypothesis. The aim of rippling is to manipulate the goal formula so that it contains a sub-formula which matches the hypothesis formula. The sub-formula can then be replaced with the formula *true*, so considerably simplifying the goal.

Rippling works by a directed rewriting of the goal. Those parts of the goal which *differ* from the hypothesis are first marked and then moved out of the way. The marked bits are called *wave-fronts*. The following analogy may help to explain what is going on.

> Imagine you are in Scotland standing beside a loch. 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 hypothesis, the reflection is the goal and the wave-fronts are the bits by which the goal differs from the hypothesis.

Figure 2 gives an example of a mathematical formula being rippled. Rippling is not just useful for inductive proofs. We have also used it, for instance, for summing series.

5 Conclusion

It is relatively easy to build a computer system for proving mathematical theorems, *in principle*. However, in practice such systems need a guidance mechanism or they will get bogged down in a combinatorial explosion. Proof planning can provide such a guidance mechanism. It requires the analysis of families of similar theorems to extract a common pattern.

With the aid of proof planning a computer system can prove theorems with very little search. This makes it more feasible to apply automated theorem proving to practical problems — in particular, to provide computer assistance for the use of formal methods. This may help make the mathematical development of software and hardware a more practical proposition, which will raise the quality of computer programs and electronic circuits.

The study of proof plans is also of interest in its own right. Proof plans describe the high-level structure of mathematical proofs and how this structure can be unpacked into the low-level details of the individual proof steps. Human mathematicians may also use proof plans to understand and produce proofs. If so, this would explain: differences in mathematical skill; how it is possible to understand a proof at a high-level, but not at a low-level, or vice versa; the

Figure 2: An Example of Rippling

Rippling is illustrated with a simple theorem about lists: the operation of appending two lists is associative, *i.e.*

$$x <> (y <> z) \quad = \quad (x <> y) <> z \tag{1}$$

where $<>$ is the infix list append operation and x, y and z are arbitrary lists. The proof is by induction on x. Assume (1) as the induction hypothesis. The induction conclusion is:

$$el :: x <> (y <> z) \quad = \quad (el :: x <> y) <> z$$

where :: is the infix operation of adding a new element to the front of a list.

Wave-fronts are expressions with holes in them. This is indicated 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. The following wave-fronts on the induction conclusion mark those bits in which it differs from the induction hypothesis.

$$\boxed{el :: \underline{x}}^{\uparrow} <> (y <> z) \quad = \quad (\boxed{el :: \underline{x}}^{\uparrow} <> y) <> z$$

To move these wave-fronts out of the way requires some wave-rules.

$$\boxed{hd :: \underline{tl}}^{\uparrow} <> l \quad \Rightarrow \quad \boxed{hd :: (\underline{tl <> l})}^{\uparrow} \tag{2}$$

$$\boxed{hd :: \underline{tl_1}}^{\uparrow} = \boxed{hd :: \underline{tl_2}}^{\uparrow} \quad \Rightarrow \quad tl_1 = tl_2 \tag{3}$$

The $\Rightarrow$ indicates that the left hand side of the rule is to be replaced by the right. The first of these wave-rules comes from the recursive definition of $<>$ as a program for appending lists. The second wave-rule comes from the definition of $=$.

Rippling of the induction conclusion can now commence.

$$\begin{aligned}
\boxed{el :: \underline{x}}^{\uparrow} <> (y <> z) \quad &= \quad (\boxed{el :: \underline{x}}^{\uparrow} <> y) <> z \\
\boxed{el :: \underline{x <> (y <> z)}}^{\uparrow} \quad &= \quad \boxed{el :: (\underline{x <> y})}^{\uparrow} <> z \\
\boxed{el :: \underline{x <> (y <> z)}}^{\uparrow} \quad &= \quad \boxed{el :: \underline{(x <> y) <> z}}^{\uparrow} \\
x <> (y <> z) \quad &= \quad (x <> y) <> z
\end{aligned}$$

The first equation is the induction conclusion annotated by wave-fronts. Subsequent equations show the effect of rippling these wave-fronts outwards using wave-rule (2) (three times) and finally wave-rule (3). 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.

difference between “standard” and “interesting” proof steps; and many similar phenomena. Proof plans enable us to analyse and categorise proofs and make possible a kind of science of reasoning.