Proving properties of logic programs: summary of progress

PROVING PROPERTIES OF LOGIC PROGRAMS: SUMMARY OF PROGRESS

Alan Bundy
Lincoln A Wallen

DAI RESEARCH PAPER NO. 312

Paper presented at Alvey Logic Programming Club,
September, 1986

Proving Properties of Logic Programs: Summary of Progress

Alan Bundy Lincoln A. Wallen

September 1986

1 Overview

The demand for software is growing to the extent that some experts predict a *software crisis*. This demand is for both more quantity and more complexity. It cannot be met using existing software development techniques both because of a shortage of skilled programmers and because of the unreliability of the software produced. To help solve this problem by improving programmer productivity and software reliability, the Alvey Directorate propose to build a knowledge-based Integrated Project Support Environment (IPSE).

Part of this knowledge-based IPSE will be systems for automatic programming, i.e. the generation of target programs from specifications, the verification that target programs meet their specifications and the transformation of inefficient, source programs into more efficient, target programs. Specifications are usually given as logical descriptions of the input/output behaviour required of the target program. If the specification is easier to write than the target program then the productivity of the programmer can be increased both because less skilful programmers can produce them and because skilled programmers can produce the target programs more quickly. If the specification or source programs are correct then the target programs are guaranteed to be correct. Thus if bug-free specifications or source programs can be written, software reliability can be improved.

Three recent developments aid the automatic programming task.

- the advent of *logic programming languages* such as Prolog, Hope and Miranda;

- the emergence of various *constructive logics* in which the notions of program and proof are fundamentally related; and

(A) Source Program (specification)   (B) Target Program

(C) Equivalence Relations

Figure 1: Property Proving Tasks in Logic Programming

- the refinement of proof guidance techniques for automatic theorem proving.

The use of logic programming and constructive logics enables the task of code generation to be viewed as theorem proving; and thus enables the new automatic proof guidance techniques to be used to guide the search.

1.1 Logic Programming

For logic programming languages the specification and the source and target program can all be written in the same language: a particular formal logic. Specifications can be viewed as possibly, very inefficient, source programs. Thus the tasks of generation, transformation and verification collapse into different aspects of the same task. Consider figure 1. The equivalence relations, (C), are the link between the source and target programs; they express one in terms of the other, e.g. by showing that both programs produce the same output.

- Program generation consists of being given (A), conjecturing (C), deriving (B) from (A) and (C), and then proving that (B) terminates. That is, given a source program, invent some more efficient target program that can be proved to be equivalent to the source program.
- Program transformation is similar except that (C) is also given.
- In program verification, (A) and (B) are given and (C) must be conjectured and derived.

The above are all examples of the proving of program properties; that is, logic programs can be thought of as formulae in a logical theory and the processes of generation, transformation and verification as theorem proving in that theory. Other kinds of program proving are possible and may be useful; for instance, we may omit (A) and only show that the target program, (B), has various desirable properties. This idea is due to Boyer and Moore, [BM73,BM79]. By regarding automatic programming as theorem proving we can apply ideas from mathematical reasoning systems to control search.

When the specification and target program languages are not the same, a program generation[1] system using the above paradigm has two tasks: to translate the specification into the programming language, and to transform the result into a more efficient program. In practice, these two tasks are interwoven. This interweaving of the two tasks greatly complicates the generation process. With logic programs only one task is necessary: the transformation of the source program into a more efficient target program. Empirical comparisons show that similar gains in efficiency can be achieved with considerably less effort in the logic programming framework.

The writing of error-free specifications/source programs in logic cannot be more difficult than writing target programs, since we need only concern ourselves with the result when writing the source program, but must also be concerned with efficiency when writing the target program. Since both programs are in the same language we do not have the overhead of thinking in two languages when setting up verification problems. Since some specifications can be run as programs they can be debugged in this way.

Thus logic programming languages facilitate automatic programming both by simplifying the task of writing bug-free specifications and by simplifying the derivations required.

1.2 Proofs as Programs

The second development concerns the use of *constructive* logics, [Mar79,C*86], to generate programs from specifications, especially specifications which cannot be run directly as programs themselves, or which define extremely inefficient programs. The technique is as follows:

- suppose that the specification defines some relationship, $f(x,y)$, between some inputs, x, and the output, y;
- form a conjecture $\forall x.\exists y.f(x,y)$, which asserts that for any inputs there is always an output satisfying the relationship;
- prove the conjecture using a constructive logic;
- extract a program from the proof.

[1]Similar remarks hold for verification.

Thus the task of building a program that satisfies a given specification reduces to the task of constructively proving a conjecture formed from that specification.

It is necessary to use constructive logic in order to avoid 'pure existence' proofs from which a program cannot be extracted. In such constructive logics we can associate the proof, a, with the formula it proves, A, to form a so called *judgement*, $a \in A$. The rules of inference for judgements define both how new formulae can be derived from old and how new proofs can be constructed from old.

But a judgement, $a \in A$, can also be read as: "a is a program for the task A". Under this interpretation the rules of inference define how new programs for new tasks can be constructed from old programs for old tasks.

It can also be read as: "a is an element of type A". Under this interpretation the rules of inference define how new objects of new types can be constructed from old objects of old types.

As an example of a rule of inference, consider the one for introduction of a bounded universal quantifier.

$$\frac{\begin{array}{c} x \in A \\ \vdots \\ b(x) \in B(x) \end{array}}{\lambda x.b(x) \ \in \ \forall x \in A.B(x)}$$

x is any proof of A. $b(x)$ is a proof of $B(x)$ which depends on x. The premise states that $B(x)$ can be proved by assuming A.

This can be read both as

- a proof rule: showing how to prove bounded universally quantified propositions, and
- a program constructing rule: showing how to form programs that realize bounded universally quantified specifications.

In a similar manner, a proof of the proposition $\exists x \in A.B(x)$ is a pair $\langle a, b(a) \rangle$ in which a is an object of type A and $b(a)$ is a proof that a satisfies the proposition $B(x)$; i.e. a proof of $B(a)$. The existential quantifier is defined by the introduction rule

$$\frac{x \in A \qquad b(x) \in B(x)}{\langle x, b(x) \rangle \ \in \ \exists x \in A.B(x)}$$

The use of a constructive logic as a language for writing specifications serves to reduce problems concerning the generation of programs to the search for proofs in the logic. Such programs are automatically verified as obeying their specification.

This technique of program generation can also be used for program transformation. The record of the rules of inference applied to generate a program contains information about the efficiency of a program which can be used to guide the theorem prover to generate an efficient program. This aspect of the proofs-as-programs paradigm has been extensively investigated by Goad at Stanford, [Goa80].

Systems that attempt to automate this form of program generation and verification have been implemented at Cornell [C*86], Goteborg [Pet82] and Stanford [MW80]. The first two using constructive type theories and the third an augmented first-order logic.

These systems are basically interactive since the proofs are well structured, but intricate. We propose to apply our ideas on search control to guide the search for such proofs.

Similar proofs and proof strategies apply here as within the logic programming paradigm since we may conjecture and prove that a target program f satisfies the specification S by conjecturing f and proving the proposition $f \in S$; i.e. f is in the class of programs specified by S. Likewise other properties of f may be proved within the same framework.

However we will need to develop new proof plans, perhaps reflecting principles of algorithm design, in order to make full use of the ability to generate programs directly from their specifications. This is a very challenging area with immense potential for the enhancement of IPSEs.

1.3 Meta-Level Inference

The problem of controlling the search for a proof has been studied extensively in the areas of automatic theorem proving and mathematical reasoning. My group has made several contributions to its solution under the collective heading of *meta-level inference*. This is the third of the recent developments mentioned above.

In meta-level inference, control information is represented as a mathematical theory in its own right; in fact, a meta-theory, because its domain of discourse consists of the formulae, terms, etc, of the theory being controlled, which is called the object-theory. The axioms of the meta-theory relate problems in the object-theory to their methods of solution. Inference in the meta-theory analyses the problem to be proved, selects an appropriate method of solution and applies it. This induces an implicit search at the object-level.

Meta-level inference was originally developed in the domain of equation solving and resulted in the PRESS program, [Bun75,BW81,SBB*82]. This work was supported by SERC grants B/SR/22993 and GR/B/73989.

Later, in an attempt to build a self-improving, equation-solving system, the technique was applied to the domain of logic programs, resulting in the IM-

PRESS program, [BS81,SB82]. In particular, IMPRESS verified logic programs which were cleaned-up versions of the Prolog code of PRESS, as part of a system for conjecturing and verifying new methods of equation solving, e.g. IMPRESS proved the *correctness of isolation*,

```
single_occurrence(X,Lhs=Rhs) &
position(X,Lhs,Posn) &
isolation(Posn,Lhs=Rhs,X=Ans)
   ⟶ solve(Lhs=Rhs,X,X=Ans)
```

which can be read as:

> If unknown X occurs only once in equation Lhs=Rhs and
> if X occurs at position Posn in Lhs and
> if isolating the X at Posn in Lhs=Rhs produces equation X=Ans
>   then X=Ans is a solution to Lhs=Rhs for X.

This work was supported by SERC grants GR/B/29252 and GR/C/20826.

IMPRESS's meta-level inference dynamically constructed an inductive proof plan to guide the search of the verification process. This proof plan was a refinement of search control ideas of Boyer and Moore, and Darlington and Burstall, [BM73,Dar81], and was also applied to logic programs drawn from their work. In particular, it extends to logic programs the use of the recursive structure of programs to guide the process of induction, and provides more guidance about how the conditions of implications might be derived, and about how different parts of a recursive definition might contribute to the proof.

When control knowledge is represented as a mathematical theory then AI techniques of deduction and learning can be applied to it. IMPRESS applied deduction to the control knowledge of PRESS, but another of our programs, LP [Sil83,Sil84], applied automatic learning. LP was input an example of an equation being solved and, using a technique called *precondition analysis*, worked out the reason behind each step of the solution. It then encapsulated this analysis in new equation solving methods and a schema which linked them, and added these to PRESS to extend the range of things it could do. It may be possible to apply the technique of precondition analysis to the domain of proving properties of logic programs.

Thus meta-level inference can guide the derivations required during automatic programming and enable more complex programs to be tackled. It also offers the prospect of automatic learning of new guidance information.

2 Original objectives

We intend to test the ideas of meta-level inference in the domain of automatic programming by building a theorem proving program incorporating the IMPRESS proof plan. This theorem prover will be tested on a wide range of problems drawn from the logic program literature, e.g. [Dar81,Cla81,Hog81,Vas85], and from the proofs-as-programs literature [Con82,Nor81]. As a result of these tests the proof plan will be refined and further proof plans developed and tested. In addition we will investigate and develop new proof plans for guiding the proofs that occur during the generation of programs from their specifications. Our target problems will be drawn from both the proof-as-programs literature cited above and various extensive case studies [MW81,Pau84].

If time allows we will also explore related issues, e.g.

- the automatic conjecture of target programs and equivalence relations (note that this involves having a theory of efficiency of programs in order to choose a target program more efficient than the source program.);
- the automatic translation of 'pure' logic programs into, for instance, Prolog code;
- the automatic acquisition of new proof plans using the ideas of precondition analysis, or other learning techniques.

3 Tools

We are using Cornell University's Nuprl [C*86] system which is an implementation of Martin-Lof's Type Theory. The Type Theory is rich enough to represent both the programs and specifications as well as the relationships that hold between them. The programming language represented in the Type Theory is a polymorphic, higher-order functional language similar to a subset of Miranda, ML or HOPE.

Proof construction operations are represented as tactics (as in LCF [GMW79]) since Nuprl contains an ML subsystem which includes a representation of the Type Theory. Tactics are functions written in ML.

4 Techniques and Progress

As mentioned before we have adopted the use of Type Theory and the Nuprl environment because it provides us with a logic for representing both programs and specifications, as well as the relationships that hold between them. Moreover the tactic framework provides us with a comprehensive language for representing proof construction operations. In the domain of automatic programming a proof plan is a specification of a series of proof construction steps, and hence program construction steps. Tactics, together with their specifications are a form of *meta-specification* of the desired proof or program structure. The flexible execution of the tactic constructs the appropriate proof (program).

Proof plans (combinations of tactics) are complex objects whose execution must be managed carefully and whose structure must be justified. It is important to be able to capture exactly when a given proof plan (or tactic) is applicable and, moreover, have some notion of what result it will produce or why it failed. To this end we are developing a suitable *tactic specification language* to enable other components of the system to reason explicitly about the utility and applicability of the proof plan and aid the automatic learning and generation of proof plans from examples. An investigation of the precondition-analysis formalism as a basis for this is underway. Preliminary results indicate that the formalism needs to be extended in order to cope with the complexity of verification proofs.

The major effort in the first part of the project has been the porting and development of the Nuprl environment. The system has been ported to the SUN workstation and the user-interface replaced with a more modular window system. The system now runs under both Suntools and Xwindows. A lot of work has been expended on this but it is considered to be of benefit to the whole UK automatic programming community. (The original Nuprl system works only on VAX and Lisp Machines.) Extended library and definition facilities have been investigated and implemented.

We have studied and refined our basic proof plan for inductive proofs and investigated its application to the existential proofs that arise when synthesizing a function from its specification. We are developing a library of tactics that may be used to develop a program and its correctness proof hand in hand.

The explicit representation of the implicit heuristics in the Boyer-Moore proof plan enables the flexible execution of the plan via meta-level inference. We are attempting to represent the justification as to why it succeeds in enough detail so that the plan may be automatically learnt from examples by means of standard explanation based learning techniques. This will open the way to automatically generalizing it in the face of problems on which it fails.

To this end we have investigated the use of various explanation based learning techniques for generalizing existing proofs and postulating new proof plans. Various case studies have been performed. Initial conclusions are that the derivations that justify programs are extremely complex and stretch the available techniques to their limit.

References

[BM73] R.S. Boyer and J.S. Moore. Proving theorems about lisp functions. In N. Nilsson, editor, *Proceedings of the third IJCAI*, pages 486–493, International Joint Conference on Artificial Intelligence, August 1973. Also available from Edinburgh as DCL memo No. 60.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[BS81] A. Bundy and L.S. Sterling. *Meta-level Inference in Algebra*. Research Paper 164, Dept. of Artificial Intelligence, Edinburgh, September 1981. Presented at the workshop on logic programming for intelligent systems, Los Angeles, 1981. A revised version is available from Edinburgh as DAI Research Paper No. 273.

[Bun75] A. Bundy. Analysing mathematical proofs (or reading between the lines). In P. Winston, editor, *Proceedings of the fourth IJCAI*, International Joint Conference on Artificial Intelligence, 1975. An expanded version is available from Edinburgh as DAI Research Report No. 2.

[BW81] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available as DAI Research Paper 121.

[C*86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Cla81] K.L. Clark. *The Synthesis and Verification of Logic Programs*. Research Report DOC 81/36, Department of Computing, Imperial College, London, September 1981.

[Con82] R.L. Constable. *Programs as Proofs*. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, November 1982.

[Dar81] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46, August 1981.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*. Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[Goa80] C.A Goad. Proofs as descriptions of computation. In W. Bibel and R. Kowalski, editors, *Proc. of the Fifth International Conference on Automated Deduction*, pages 39–52, Springer Verlag, Les Arcs, France, July 1980. Lecture Notes in Computer Science No. 87.

[Hog81] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[Mar79] Per Martin-Lõf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[MW81] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[Nor81] B. Nordstrom. Programming in constructive set theory: some examples. In *Proceedings of the ACM conference on Functional Programming and Computer Architecture*, pages 141–153, ACM, Porstmouth, NH, October 1981.

[Pau84] L. Paulson. *Verifying the Unification Algorithm in LCF.* Report 50, Computer Laboratory, University of Cambridge, 1984.

[Pet82] K. Petersson. *A Programming System for Type Theory.* Memo 21, Chalmers University of Technology, 1982. Programming Methodology Group.

[SB82] L. Sterling and A. Bundy. Meta-level inference and program verification. In D.W. Loveland, editor, *6th Conference on Automated Deduction*, pages 144–150, Springer Verlag, 1982. Lecture Notes in Computer Science No. 138. Also available from Edinburgh as Research Paper 168.

[SBB*82] L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, and B. Silver. Solving symbolic equations with press. In J. Calmet, editor, *Computer Algebra, Lecture Notes in Computer Science No. 144.*, pages 109–116, Springer Verlag, 1982. Longer version available from Edinburgh as Research Paper 171. To appear in Journal of Symbolic Computation.

[Sil83] B. Silver. Learning equation solving methods from examples. In A. Bundy, editor, *Proceedings of the Eighth IJCAI*, pages 429–431, International Joint Conference on Artificial Intelligence, 1983. Also available from Edinburgh as Research Paper 184.

[Sil84] B. Silver. Precondition analysis: learning control information. In *Machine Learning 2*, Tioga Publishing Company, 1984.

[Vas85] P.E. Vasey. *First-Order Logic Applied to the Description and Derivation of Programs.* PhD thesis, Imperial College, 1985.