The use of proof plans in formal methods

The Use of Proof Plans in Formal Methods

Alan Bundy
Department of Artificial Intelligence,
University of Edimburg, Scotland

1. Introduction

Proof plans is a new AI technique for controlling the search that arises in automatic theorem proving. We have applied it to the satisfaction of the proof obligations arising from the use of formal methods in software engineering. We first describe the particular formal methods technique we have adopted as a vehicle and then describe how proof plans are used within this technique.

2. Program Synthesis using Theorem Proving

We are concerned with the synthesis of logic/functional programs in the Nuprl style, [Constable et al 86]. The basic idea is to start with a logical specification, *spec(Inputs, Output)* between the inputs to and outputs from a program, and then prove a conjecture of the form:

$$\forall \mathit{Inputs}, \exists \mathit{Output.spec}(\mathit{Input},\mathit{Output})$$

in a constructive logic. Because a constructive logic is used, any proof of this conjecture must implicity encode a program, prog(Inputs), which obeys the specifications, i.e. for which:

$$\forall \mathit{Inputs.spec}(\mathit{Input},\mathit{prog}(\mathit{Input}))$$

Nuprl uses a logic based on Martin Lof Intuitionist Type Theory, [Martin-Lof 79]. This makes trivial the extraction of prog from the proof, since each rule of inference of the logic has an associated program construction step. The program is, thus, built as a side effect of constructing the proof. There is a direct relation between each proof step and the corresponding part of the program, for instance, proofs by mathematical induction create recursive programs. We are, therefore, particularly interested in inductive proofs. The program is a logic/functional program in the Type Theory logic. As a programming language, this logic is higher order with very flexible types. Type checking is done at synthesis time.

Naturally the theorem proving required to do this synthesis is combinatorially explosive - in fact, the Type Theory is more badly behaved in this respect than resolution theorem provers. For instance, it has a potentially infinite set of rules of inference, some of which have infinite branching rates. It is an open question as to whether this bad behaviour can be tamed with throwing away the benefits of the logic from a program synthesis point of view. The Nuprl solution to this problem is to control the search by a combination of user interaction and built-in simplification routines. The latter are implemented as tactics: ML programs which call various rules of inference when executed, cf. LCF. The user can also use custom built tactics to encode a sequence of rule applications.

We have built our own version of Nuprl, which we call Oyster, [Horn 88]. It differs from Nuprl in being implemented in Prolog rather than Lisp, being considerably smaller and cheaper to run, and using Prolog rather than ML as the tactic language. We have found that the pattern directed invocation of Prolog makes the writing of tactics much simpler and clearer than with ML.

3. Proof Plans to Control Search

Our work has been to try to automate the search process to a much greater extent than in Nuprl or similar systems. We have adapted the inductive proof heuristics of Boyer and Moore, [Boyer & Moore 79], to the Oyster system, and implemented them as tactics. These tactics have been successfully tested on a number of standard theorems from the literature, [Bundy et al 89a].

In Boyer and Moore's system, their heuristics are applied in a fixed order. This makes their system brittle. We have been developing a technique, called proof plans, for applying the tactics in a more flexible manner. Each tactic is partially specified in a method. Our Clam plan formation program, [van Harmelen 89], is then used to build a proof plan especially adapted to the current conjecture. This has give our system improved performance over the Boyer-Moore system.

We have also analysed the Boyer-Moore heuristics and rationally reconstructed the reasoning behind their design and order. This analysis has been captured in a tactic, the induction strategy, which is at a higher level of abstraction than any of their heurisitcs, and is extremely successful in proving theorems. the analysis has also enabled us to generalise some of the heuristics and add new ones, extending the power of the system. For instance we can now prove existential theorems (Boyer and Moore are restricted to universal quantification) and use inductive schenmnata that are not suggested by recursions in the original conjecture, [Bundy et al 89b].

References

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. A Computational Logic. Academic Press, 1979.ACM monograph series.

[Bundy et al 89a] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Journal of Automated Reasoning, 1989. In press. Earlier version available from Edinburgh as Research Paper No 413.

[Bundy et al 89b] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, pages 359-365, Morgan Kaufmann, 1989. Available from Edinburgh as Research Paper 419.

[Constable et al 86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. Implementing Mathematics with the Nuprl Proof Development System., Prentice Hall, 1986.

[Horn 88] C.Horn. The Nuprl Proof Development System. Working Edinburgh version of Nuprl has been renamed Oyster.

[Martin-Lof 79] Per Martin-Lof. Constructive mathematics and computer programming. In 6th International Congress for Logic, Methodology and Philosophy of Science, pages 153-175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[van Harmelen 89] F. van Harmelen. The CLAM Proof Planner, User Manual and Programmer Manual. Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.