Automatic guidance of program synthesis proofs

Automatic Guidance of Program Synthesis Proofs [*]

Alan Bundy.
Department of Artificial Intelligence,
University of Edinburgh,
80 South Bridge,
Edinburgh, EH1 1HN,
Scotland.
Email: bundy@edinburgh.ac.uk

1 Program Synthesis using Theorem Proving

A technique called *proof plans* will be described, together with its application to the synthesis of logic/functional programs in the Nuprl style, [Constable *et al.*, 1986].

Nuprl-style program synthesis requires the proving of inductive theorems. Proof plans are used to control the search that arises from the automation of this theorem proving. The basic idea of Nuprl-style synthesis 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:

$$\forall Inputs, \exists Output.spec(Inputs, Output) \qquad (1)$$

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

$$\forall Inputs.spec(Inputs, prog(Inputs))$$

Nuprl uses a logic based on Martin Lõf Intuitionist Type Theory, [Martin-Lõf, 1979]. 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. 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. 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, 1988]. It differs from Nuprl in being implemented in Prolog rather than Lisp, being considerably smaller and more transparent, 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.

2 Proof Plans to Control Search

Our work has been to try to automate the search process to a much greater extent. We have adapted the inductive proof heuristics of Boyer and Moore, [Boyer and Moore, 1979], 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.*, 1988a, Bundy *et al.*, 1988b].

In Boyer and Moore's system, their heuristics are applied in a fixed order. This makes their system brittle. *Proof plans* permit a flexible application of tactics. Each tactic is partially specified in a *method*. This method gives preconditions for the use of the tactic and describes the effect of using it. These preconditions and effects are written in a meta-logic, *i.e.* a logic for reasoning about the representation in which the conjectures are expressed. Our Clam system, [van Harmelen, 1989], uses AI plan formation techniques to reason with these methods to build a proof plan especially adapted to the current conjecture. This has given our system improved performance over the Boyer-Moore system, *e.g.* we can prove the commutativity of multiplication without the need for prestored lemmas.

Figure 1 is a simple example of the kind of proof generated by the BMTP and by our proof plans. It illustrates the tactics used to control Oyster and the way in which Clam puts these tactics together into a proof plan. The theorem is that the sum of two even numbers is even. The notation is based on that used by Oyster, but it has been simplified for expository reasons and only the major steps of the proof have been given.

Each formula is a sequent of the form $H \vdash G$, where H is a list of hypotheses and G is a goal. Formulae of the form $X : T$ are to be read as "X is of type T".

[*] The research reported in this extended abstract was supported by SERC grant GR/E/44598, Alvey/SERC grant GR/D/44270 and an SERC Senior Fellowship to the author. It has been conducted my members of the Mathematical Reasoning Group over several years.

pnat is the type of Peano natural numbers. The first sequent is a statement of the theorem. Its first five hypotheses constitute the recursive definitions of + and *even*. Each subsequent sequent is obtained by rewriting some subexpressions in the one above it. A subexpression to be rewritten is underlined and the subexpression which replaces it is overlined and connected to it with an arrow. Only newly introduced hypotheses are actually written in subsequent sequents; subsequent sequents are to be understood as inheriting all those hypotheses above them in the proof. In the spaces between the sequents are the names of the tactics which invoke the rewriting.

The proof in figure 1 is by backwards reasoning from the statement of the theorem. The *induction* tactic applies a two step induction rule to the theorem: replacing x by 0 in the first base case, s(0) in the second and by $s(s(x'))$ in the induction conclusion of the step case. The *base* and *wave* tactics then rewrite the base cases and the step case, respectively, using the base and step equations of the recursive definition of + and *even*. The applications of *base* rewrite the base cases to propositional tautologies, which the *sym_eval* tactic reduces to $\vdash$ *true*. The four applications of *wave* raise the occurrences of the successor function, s, from their innermost positions around the x's until they are absorbed by applications of the *even* wave rule. We call this process *rippling-out*. The induction conclusion is then identical to the induction hypothesis, so the *fertilize* tactic can prove the former from the latter. The *ind_strat* is a tactic for guiding the whole of this proof, apart from the two *sym_eval* steps. It is defined by combining the subtactics *induction*, *base*, *wave* and *fertilize* in the order suggested by the proof in figure 1.

The *wave* tactic works by applying rewrite rules of the form:

$$F(\underbrace{S_1(U_1)},\ldots,\underbrace{S_n(U_n)}) \quad \Rightarrow \quad T(\overbrace{F(U_1,\ldots,U_n)})$$

which we call *wave rules*. The terms marked by an underbrace are called *wave terms*. The terms marked by an overbrace are called *wave functions*. The idea of wave rules is to ripple wave terms out from inside wave functions to outside wave functions. In the process the wave terms usually change to some other term, possibly disappearing altogether. Examples of wave rules are:

$$\underbrace{s(u_1)} = \underbrace{s(u_2)} \quad \Rightarrow \quad \overbrace{u_1 = u_2}$$

$$\underbrace{s(u)} + v \quad \Rightarrow \quad s(\overbrace{u+v})$$

$$even(\underbrace{s(s(u))}) \quad \Rightarrow \quad \overbrace{even(u)}$$

Note that the step cases of the recursive definitions of + and *even* provide wave rules, but that wave rules are not limited to recursive definitions.

The preconditions of the method for *ind_strat* look ahead to the ripple out process and the wave rules that are available to conduct it. They use this analysis to suggest a form of induction that is likely to lead to a successful ripple out (for details see [Bundy *et al.*, 1988b]). Expressed in our meta-logic, these preconditions are:

$$\begin{array}{l}
occ(Conj, X) \wedge \\
\forall Posn' : lists(pnat), \forall N' : pnat. \\
\quad \{exp\_at(Conj, [N'|Posn']) \equiv X \longrightarrow \\
\quad \exists WaveFunc' : terms, \exists WaveTerm' : terms. \\
\qquad \{exp\_at(Conj, Posn') \equiv WaveFunc' \wedge \\
\qquad\quad wave\_rule(WaveFunc', N', WaveTerm', Rule) \wedge \\
\qquad\quad subsumes(IndTerm, WaveTerm')\}\}
\end{array}$$

where *Conj* is the conjecture, X is the induction variable and *IndTerm* is the *induction term* of an induction rule of inference. The induction term of a rule is the term which it substitutes for the induction variable in the conjecture to form the induction conclusion, *e.g.* $s(s(x'))$ in figure 1. Note that the preconditions of *ind_strat* are expressed in the same underlying type theory as the object-level formulae, but we adopt the convention of using capitalised words as meta-level variables and lower case words for meta-level constants and all object-level symbols.

The declarative reading of these preconditions is:

- that X occurs in *Conj*;
- that each occurrence of X is in the wave argument position of a function for which there is a matching wave rule; and
- that there is a term, *IndTerm*, that subsumes each of the wave terms, *WaveTerm'*, of these wave rules.

These preconditions ensure that if an induction rule with induction term, *IndTerm*, is used then at least the first round of rippling out will succeed. They are usually checked with *Conj* instantiated to some specific conjecture, but X and *IndTerm* uninstantiated. The process of checking instantiates X and *IndTerm* to some values for which the preconditions are true — effectively analysing the conjecture to suggest a form of induction that will make rippling out likely to succeed. In practice, the checking process is flexible so that partial success is accepted when total success is not possible. For instance, if the conjecture is $x + y = y + x$ then no choice of induction variable fully meets the preconditions, but x/X and $s(x)/IndTerm$ is accepted as one of the two best available alternatives.

We have analysed the Boyer-Moore heuristics and rationally reconstructed the reasoning behind their design and order. This analysis has been captured in the *ind_strat* and *sym_eval* tactics, which are at a higher level of abstraction than any of their heuristics, and are extremely successful in proving theorems. The analysis has also enabled us to generalise some of their 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 schemata that are not suggested by any recursions in the original conjecture, [Bundy *et al.*, 1988b]. These two abilities are vital for our chosen application of program synthesis. Firstly, we must be able to prove existential theorems of the form outlined in formula (1) above. Secondly, the program specifications should not have to encode algorithmic information such as that implicitly represented in a recursive defini-

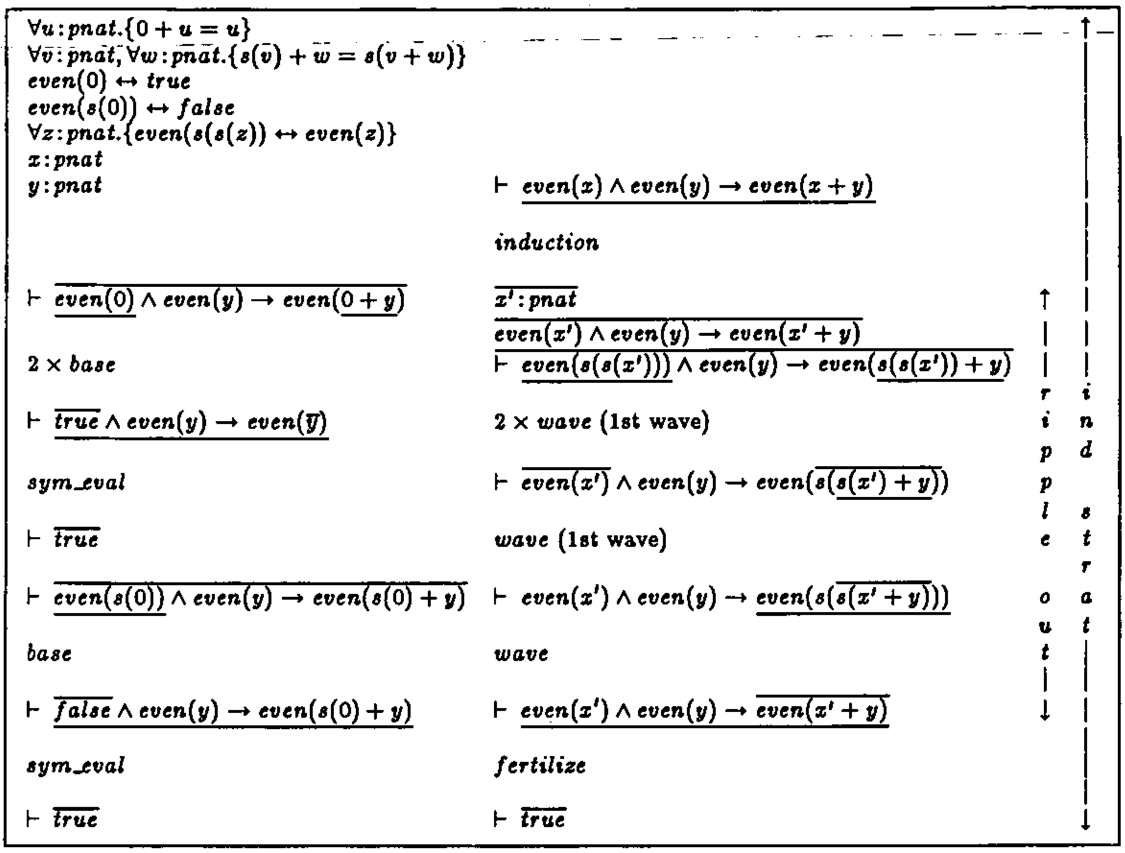
$\forall u{:}pnat.\{0 + u = u\}$
$\forall v{:}pnat, \forall w{:}pnat.\{s(v) + w = s(v + w)\}$
$even(0) \leftrightarrow true$
$even(s(0)) \leftrightarrow false$
$\forall z{:}pnat.\{even(s(s(z)) \leftrightarrow even(z)\}$
$x{:}pnat$
$y{:}pnat$
$\vdash even(x) \wedge even(y) \rightarrow even(x + y)$
induction
$\vdash even(0) \wedge even(y) \rightarrow even(0 + y)$
2 × base
$\vdash true \wedge even(y) \rightarrow even(y)$
sym_eval
$\vdash true$
$\vdash even(s(0)) \wedge even(y) \rightarrow even(s(0) + y)$
base
$\vdash false \wedge even(y) \rightarrow even(s(0) + y)$
sym_eval
$\vdash true$
$x'{:}pnat$
$even(x') \wedge even(y) \rightarrow even(x' + y)$
$\vdash even(s(s(x'))) \wedge even(y) \rightarrow even(s(s(x')) + y)$
2 × wave (1st wave)
$\vdash even(x') \wedge even(y) \rightarrow even(s(s(x') + y))$
wave (1st wave)
$\vdash even(x') \wedge even(y) \rightarrow even(s(s(x' + y)))$
wave
$\vdash even(x') \wedge even(y) \rightarrow even(x' + y)$
fertilize
$\vdash true$
ripple out
ind strat

Figure 1: Outline Proof of the Even+ Theorem

tion. We want recursion-free specifications to give rise to recursive programs. This is now possible in our system.

References

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

[Bundy *et al.*, 1988a] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, 1988. Submitted to JAR.

[Bundy *et al.*, 1988b] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. Research Paper 419, Dept. of Artificial Intelligence, Edinburgh, 1988. To appear in the proceedings of IJCAI-89.

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

[Horn, 1988] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Martin-Löf, 1979] Per Martin-Löf. 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.

[van Harmelen, 1989] F. van Harmelen. The CLAM proof planner, user manual and programmer manual. Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.