Experiments with Proof Plans for Induction

Alan Bundy Frank van Harmelen Jane Hesketh
Alan Smaill
Department of Artificial Intelligence,
University of Edinburgh.

October 28, 1988

Abstract

The technique of *proof plans*, is outlined. This technique is used to guide automatic inference in order to avoid a combinatorial explosion. Empirical research to test this technique in the domain of theorem proving by mathematical induction is described. Heuristics, adapted from the work of Boyer and Moore, have been implemented as Prolog programs, called *tactics*, and used to guide an inductive proof checker, *Oyster*. These tactics have been partially specified in a meta-logic, and plan formation has been used to reason with these specifications and form plans. These plans are then executed by running their associated tactics and, hence, performing an Oyster proof. Results are presented of the use of this technique on a number of standard theorems from the literature. Searching in the planning space is shown to be considerably cheaper than searching directly in Oyster's search space. The success rate on the standard theorems is high. These preliminary results are very encouraging.

Keywords

Theorem proving, mathematical induction, search, combinatorial explosion, proof plans, tactics, planning.

Acknowledgements

The research reported in this paper was supported by SERC grant GR/E/44598, Alvey/SERC grant GR/D/44270, and an SERC Senior Fellowship to the first author. We are grateful to the other members of the mathematical reasoning group at Edinburgh for many useful discussions.

1 Introduction

This paper describes work in progress to explore the use of *proof plans* for the automatic guidance of proofs by mathematical induction.

Such inductive proofs are required in the domain of verification, transformation and synthesis of recursive computer programs. We have adopted this domain as a vehicle for the exploration of our ideas on automatic guidance. To enable us to do this the Nuprl program development system, [Constable *et al* 86], has been reimplemented in Prolog by Christian Horn, a visitor to our group, [Horn 88]. This system, which we have christened, *Oyster*, is a proof checker for Intuitionistic Type Theory, based on a system of Martin-Löf, [Martin-Löf 79]. This is a constructive, higher order, typed logic, especially suitable for the task of program synthesis.

Oyster reasons backwards from the theorem to be proved using a sequent calculus notation, which includes rules of inference for mathematical induction. The search for a proof must be guided either by a human user or by a Prolog program called a *tactic*. The Oyster search space is very big, even by theorem proving standards. There are hundreds of rules of inference, many of which have an infinite branching rate. So careful search is very important if a combinatorial explosion is to be avoided. Most of these huge Oyster search spaces consist of sub-proofs that various expressions are well typed. These provide a synthesis time type checking on the programs synthesised by the proofs. These sub-proofs are fairly easy to control, but even without them the search spaces are very big. It is an open problem whether the usual devices of normal forms, unification, *etc.* can be used to make a more computationally tractable theorem prover without sacrificing its suitability for program synthesis.

Our aim is to develop a collection of powerful, heuristic tactics that will guide as much of the search for a proof as possible, thus relieving the human user of a tedious and complex burden. These tactics need to be applied flexibly in order to maximise Oyster's chances of proving each theorem.

The state of the art in inductive theorem proving is the Boyer-Moore Theorem Prover, [Boyer & Moore 79] (henceforth BMTP). It is, thus, natural for us to try and represent the heuristics embedded in the BMTP as Oyster tactics. [Bundy 88] contains an analysis of some of these heuristics. We have used this analysis to implement a number of Oyster tactics for inductive proof and have tested them on some simple theorems, in the theories of natural numbers and lists, drawn from [Boyer & Moore 79] and [Kanamori & Fujita 86]. These tactics are outlined in §2.

A theorem prover faithful to the spirit of BMTP would apply these tactics, in sequence, to a series of sequents. It would use a process of backwards reasoning:

with the theorem to be proved as the initial sequent and a list of axiom sequents, *true*, as the final one. Whenever a tactic succeeded in modifying the current sequent, the resulting formula would become the new sequent and would be sent to the beginning of the tactic sequence. If the current sequent could not be modified then the theorem prover would fail rather than backtrack. The BMTP does not search.

Clearly this strategy is very reliant on the design of the tactics and on their order of application. We were keen to improve on this strategy by making the tactic application order more sensitive to the theorem to be proved and, hence, less reliant on the tactic design. We have built a number of plan formation programs which construct a *proof plan* consisting of a tree of tactics customised to the current theorem, and have tested these planners on our standard list of theorems. These planners are described in §3.

In order to build this plan it is necessary to specify each tactic, partially, by giving some preconditions for its attempted application and some effects of its successful application. We call this partial specification a *method*. It is expressed in a *meta-logic*, whose domain of discourse consists of logical expressions and tactics for manipulating them. More details of the advantages and use of proof plans can be found in [Bundy 88].

2 Tactics for Guiding Inductive Proofs

Figure 1 is a simple illustrative example of the kind of proof generated by BMTP and by our Oyster tactics. It is the associativity of + over the natural numbers. 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, $\vdash$ is the sequent arrow and G is a goal. Formulae of the form $X : T$ are to be read as "X is of type T". *pnat* is the type of Peano natural numbers. The first sequent is a statement of the theorem. Its first two hypotheses constitute the recursive definition of +. Each subsequent sequent is obtained by rewriting a subexpression in the one above it. The subexpression to be rewritten is underlined and the subexpression which replaces it is overlined. Only newly introduced hypotheses are actually written in subsequent sequents; they are to be understood as inheriting 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 is by backwards reasoning from the statement of the theorem. The *induction* tactic applies the standard arithmetic induction schema to the theorem: replacing x by 0 in the base case and by $s(x')$ in the induction conclusion of the step case. The *take_out* and *unfold* tactics then rewrite the base and step

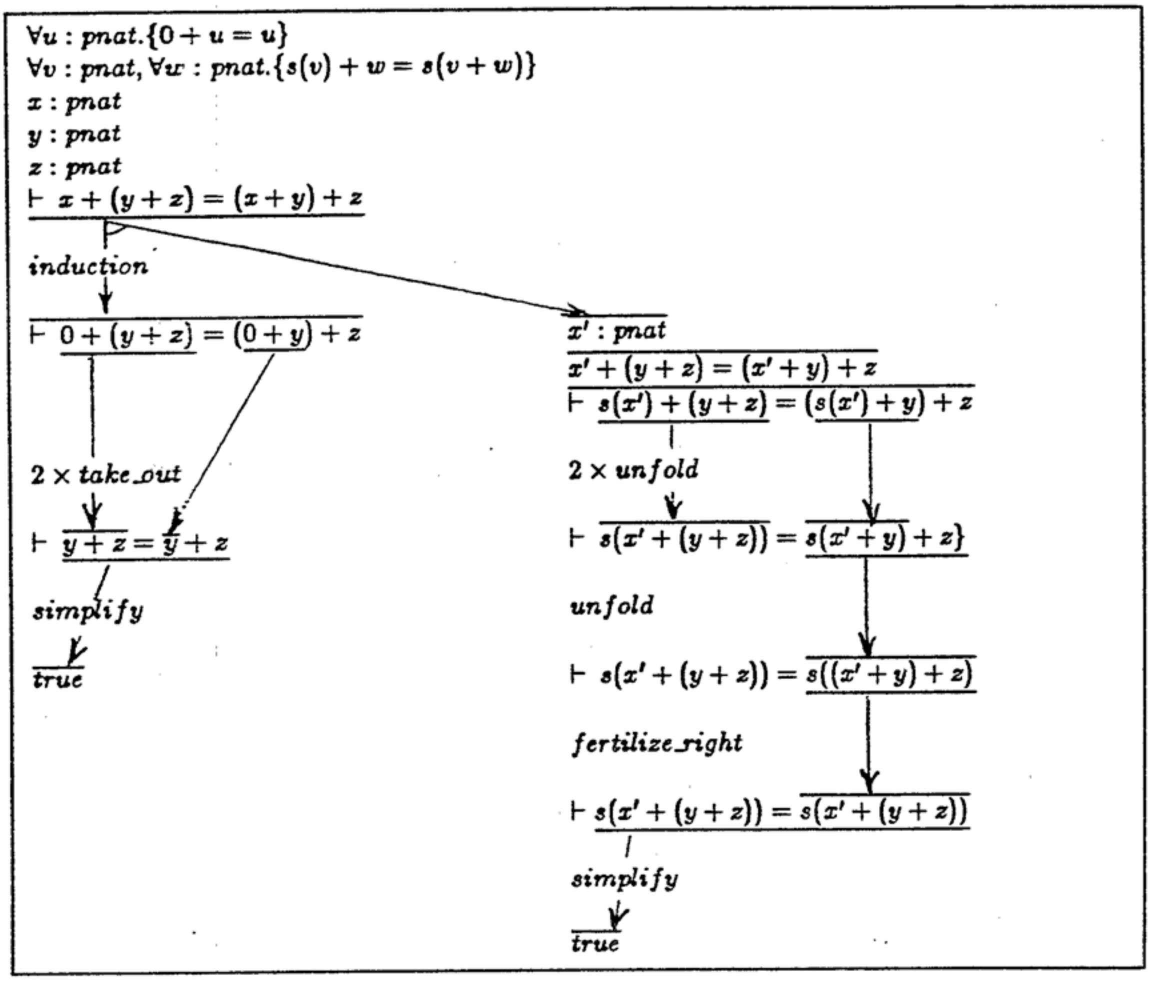

Figure 1: Outline Proof of the Associativity of +

case, respectively, using the base and step equations of the recursive definition of +. The two applications of *take_out* rewrite the base case to an equation between two identical expressions, which the *simplify* tactic reduces to *true*. The three applications of *unfold* raise the occurrences of the successor function, s, from their innermost positions around the x's to being the outermost functions of the induction conclusion. The two arguments of the successor functions are then identical to the two arguments of = in the induction hypothesis. The *fertilize_right* tactic then replaces the right hand of these two arguments for the left hand one in the induction conclusion. The two arguments of the successor functions are now identical and the *simplify* tactic reduces the sequent to *true*. The *basic_plan* is a tactic for guiding the whole of this proof, apart from the two *simplify* steps. It is defined by combining the sub-tactics *induction*, *take_out*, *unfold* and *fertilize_right* in the order suggested by the above proof.

Each of these tactics is implemented as a Prolog program that calls Oyster rules of inference in order to manipulate the current sequent and produce a new one. As an example, the *unfold* tactic is given in figure 2. The argument to the *unfold*/1 procedure, (0), is the position of the constructor function we want to unfold. This position is represented as a list of numbers, *e.g.* [1,2,3] represents the 1st argument of the 2nd argument of the 3rd argument of the outermost function. *unfold*/1 first picks up the current sequent, (1), and finds the subexpression containing the specified occurrence of the constructor, (2). It finds the function symbol whose step-equation it wants to use, (3), and picks up its step-equation, (4). It then finds the new value of the subexpression-to-rewrite by computing the instantiation of the step equation that matches the subexpression-to-rewrite, (5), and uses that result to call the sub-tactic *rewrite*, (6). If this succeeds then it produces a list of three sequents, the first and the third one of which are proved by the sub-tactics *univ_elim* and *wfftacs*, and the second one of which is left as the remaining subgoal[1] after the application of *unfold*, (6).

```
unfold([_|Pos]) :-                                          (0)
        goal(G),                                            (1)
        exp_at(G, Pos, Exp),                                (2)
        exp_at(Exp, [0], F),                                (3)
        step(F, Eq),                                        (4)
        instantiate(Eq, Exp = NewExp in T, _),              (5)
        rewrite(Exp = NewExp in T)
                then [univ_elim, idtac, wfftacs].           (6)
```

Figure 2: Prolog Code for the *unfold* Tactic

A selected list of theorems to which these tactics have been applied is given in figure 1. The cpu times taken to prove these theorems and the lengths of the proofs found are tabulated in table 2, in columns OT and OL, respectively.

[1] *idtac* is a non-op.

3 Using Planning for Flexible Application

We have had some success in proving theorems by repeated application of the *basic_plan* and *simplify* tactics. This success confirms the hypothesis proposed in [Bundy 88] that the proof structure captured in *basic_plan* underlies a large number of inductive proofs. However, some theorems (*e.g.* *com*×) do not yield to this straightforward combination of tactics and require *ad hoc* modifications, *e.g.* using *take_out* in the step case. This kind of *ad hoc* patching is unlikely to work for more complex theorems. To make a powerful theorem prover which will scale up to complex theorems, it is necessary to put the tactics together in a principled and flexible way. That is, we want tactics used to be sensitive to the form of theorem to be proved and to be explicable in terms of that form.

To achieve this we use AI plan formation techniques to construct super-tactics, especially geared to the theorems to be proved, out of the sub-tactics described above. Each of these sub-tactics is partially specified using a *method* and the plan formation program reasons with these methods to link the sub-tactics together. Example plans formed by this process are given in figures 5 and 6. The theorems are then proved by executing the super-tactics defined by these plans.

A method is represented as an assertion of the Prolog procedure *method*/6 in the format given in figure 3. The first argument, (1), to *method*/6 is the name of the method: a function with some arguments specifying the context of its use. We find it convenient, in practice, to overload the tactic name and reuse it as the method name. The second argument, (2), is the *input formula*, a meta-level pattern which any formula input to the tactic must match. The third argument, (3), is the *preconditions*, a list of further properties, written in the meta-logic, that the input formula must satisfy. The fifth argument, (5), is the *output formulae*, a list of meta-level patterns which any formulae output by the tactic will match. The fourth argument, (4), is the *effects*, a list of further properties, written in the meta-logic, that the output formula will satisfy. The sixth argument, (6), is the Prolog procedure call to the tactic.

The method for the *unfold* tactic is given, as an example, in figure 4. The input to the tactic, (2), can be any sequent, $H \vdash G$, where H is the hypothesis and G is the goal. The argument, $[N|Pos]$, to the name, *unfold*, of the method, (1), and the tactic, (6), is a list of numbers specifying a position. The preconditions, (3), for attempting the tactic are as follows. In position $[N|Pos]$ in G there should be a constructor term, *Constructor* with a constructor function, *ConstructorFunc* as its dominant function. *Constructor* should be in the recursive argument position of a primitive recursive function, F, whose recursive definition has the step case, *StepEq*. The result of a successful application of the tactic will be that the output, (5), will be a sequent $H \vdash NewG$, in which $NewG$

$$
\begin{array}{ll}
method(name(\ldots Args \ldots), & (1)\\
\quad Input\ formula, & (2)\\
\quad Preconditions, & (3)\\
\quad Effects, & (4)\\
\quad Output\ formulae, & (5)\\
\quad tactic(\ldots Args \ldots) & (6)\\
\quad). &
\end{array}
$$

Figure 3: The Format of Methods

is formed from G by rewriting the term at position Pos using $StepEq$, (4).

Finding proof plans presents an unusual plan formation problem. Most AI planners work backwards from the final goal[2] to the initial state. Unfortunately, the final goal of all our proofs is a list of *trues*, and this gives the planner virtually nothing to work from. The initial state, *i.e.* the theorem to be proved, is a much richer source of information. Therefore, we have built a series of experimental *forward* planners.

Altogether we have built four different forward planners. Our *depth-first* planner is the fastest at finding plans, but sometimes gets trapped down an infinite branch of the planning search space and does not always find the shortest plan. Our *breadth-first* planner is guaranteed to terminate with the shortest plan, if there is a plan, but is intolerably slow on all but trivial theorems. Our *iterative deepening* planner is a fairly good compromise, being much faster than the breadth-first one and being guaranteed to terminate with the shortest plan. Our *best-first* planner is only slightly slower than the depth-first planner and, in practice, usually terminates with plans of reasonable length. Its heuristics consist of a simple fixed order in which to try the methods.

Each planner takes the theorem to be proved as the initial state and finds a tree of methods which will transform it into a list of *trues*. At each cycle it finds a method that is applicable to the current state by matching that state to the input pattern of the method and checking the preconditions. The list of output formulae is then calculated from the output and the effects of the method. The cycle is repeated for each of these output formulae.

[2]Note that goals in planning are not the same thing as goals in sequents.

$$
\begin{array}{ll}
method(unfold([N|Pos]), & (1)\\
\quad H \vdash G, & (2)\\
\quad [\ type(\_,\_,\_,Constructor), & \\
\quad\quad exp\_at(Constructor,[0],ConstructorFunc), & \\
\quad\quad exp\_at(G,[0,N|Pos],ConstructorFunc), & \\
\quad\quad exp\_at(G,[0|Pos],F), & \\
\quad\quad prim\_rec(F,N), & \\
\quad\quad step(F,StepEq) & \\
\quad], & (3)\\
\quad [rewrite(Pos,StepEq,G,NewG)], & (4)\\
\quad [H \vdash NewG], & (5)\\
\quad unfold([N|Pos]) & (6)\\
\quad). &
\end{array}
$$

Figure 4: The Method for the *unfold* Tactic

For instance, if the current state were the sequent:

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

then the method $unfold([1,1,1])$ is applicable since there is a constructor term, $s(x)$, in position $[1,1,1]$ in the sequent's goal, in the recursive argument position of a primitive recursive function $+$. After rewriting the term in position [1,1] with the step case of the recursive definition of $+$ we get the output sequent:

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

When the tactic $unfold([1,1,1])$ is executed it generates an Oyster proof consisting of 25 rule of inference applications! This 25:1 ratio indicates the gearing that we get from planning the proof. Further evidence for this can be found in table 2. 22 of these are concerned with proving well-typedness, but even if these are ignored the remaining 3:1 ratio still indicates a significant gearing.

If the *basic_plan* method is not available, the plan found for the example *ass+* is as displayed in figure 5. When the tactic corresponding to this plan is executed it generates the proof outlined in figure 1, as required. If the *basic_plan* method is available, the plan found is as displayed on the left hand side of figure 6. Of course, the tactic associated with this plan also generates the proof outlined in figure 1. The right hand side of figure 6 shows the plan formed for the example *com+*. This illustrates the way in which the *basic_plan* can be nested in a plan.

```
induction(x) then
             [ take_out ([1,1,1]) then
                     take_out ([1,1,2,1]) then
                              simplify ,
              unfold([1,1,1]) then
                     unfold([1,1,2,1]) then
                              unfold([1,2,1]) then
                                        fertilize_right([1],v3) then
                                                  simplify
             ]
```

Figure 5: The Proof Plan Generated for *ass+*

```
basic_plan(x) then          basic_plan(x) then
      [ simplify,                 [ basic_plan(y) then
        simplify                          [ simplify,
      ]                                     simplify
                                          ],
                                    basic_plan(y) then
                                          [simplify,
                                            simplify
                                          ]
                                  ]
```

Figure 6: The Plans for *ass+* and *com+* using the *basic_plan* Method

4 Results

The results of applying our plan formation programs to the theorems listed in table 1 and then executing the resulting plans in Oyster, are given in table 2. The meaning of the various columns is as follows.

- PT — is the time in cpu seconds to form the plan using the best-first planner. All cpu times were measured using a Sun3/60 with 24 Mb of memory, running Quintus 2.2 under SunOS 3.5. A "-" sign indicates that the attempt to find a plan failed. With the depth first planner times are slightly shorter, but fewer planning attempts are successful. With the iterative deepening planner times are slightly longer and exactly the same planning attempts succeed. With the breadth first planner times are several orders of magnitude longer and many planning attempts had to be abandoned due to resource limitations.

- OT — is the time in cpu seconds to execute the plan by running its associated Oyster tactic. This calls rules of inference of Martin-Lōf Intuitionistic Type Theory.

- RT — is the result of dividing OT by PT. These results were very surprising to us. It is an order of magnitude less expensive to find a plan than to execute it, despite that fact that finding a plan involves search whereas executing it does not. Partly this is due to an inefficient implementation of the application of Oyster rules of inference. However, it also reflects the smaller length of plans compared to proofs, the small size of the plan search space (*cf.* column PS) and the inherent cheapness of calculating method preconditions and effects. It also indicates that most of the time spent executing a tactic is taken up in applying Oyster rules of inference, rather than in locating the rule to apply.

- PL — is the length of the shortest plan found by the best-first planner, *i.e.* the number of tactics in the plan. These are calculated with the *basic_plan* tactic available.

- OL — is the length of the proof found by executing tactics corresponding to the plan, *i.e.* the number of applications of Oyster's rules of inference in the proof.
  Note to referee: The figures given (except one) are estimates. Measurement of the precise figures requires a modification to the Oyster system, which we were not able to implement before the deadline, but the figures will be available shortly.

- RL — is the result of dividing OL by PL. Note that plans are significantly shorter than proofs. This is because each tactic applies several rules of inference.

- PS — is the number of nodes visited in the planning space before the first plan is found by the iterative deepening planner. For some of the simpler theorems we give figures both with and without the *basic_plan*. The former are much smaller, since short plans using *basic_plan* are found early in the search. For more difficult theorems only the figure with the *basic_plan* is given, although our planners can find plans *not* containing the $basic_plan$ for all those theorems for which they can find plans containing the $basic_plan$. Resource limitations prevented the iterative deepening planner finding a plan for some theorems, even though the best-first planner had succeeded. We have estimated PS in these cases.

- OS — is an estimate of the number of nodes visited in the object-level space before the first proof is found by the iterative deepening planner. Those rules that generate infinite branching points were restricted in application to a finite number of sensible instances. Attempts to automate even this restricted version ran into severe resource problems due to the huge size of the object-level search space, so an estimate had to be made.

- RS — is the result of dividing OS by PS. This shows the considerably smaller size of the plan search compared to the proof search space. We used the same iterative deepening planner for calculating/estimating PS and OS, in order to facilitate comparison. We rejected the best-first planner for this purpose because it would have been necessary to provide different heuristics for the plan and object-level searches, thus obscuring the comparison.

These initial results are very encouraging. The much smaller search space required for planning as opposed to theorem proving (see column RS) shows a considerable potential for defeating the combinatorial explosion by finding plans and then executing them, rather than searching for proofs directly. We do not have to pay for this decrease in search space by an increased cost of searching. On the contrary, column RS shows that it is considerably *cheaper* to search in the planning space. The relatively high cost of executing the plan would need to be paid anyway during the search of the object-level search space, since most of the run time of a tactic is spent in applying rules. In fact, much more would have to be paid, since it would cost more to search for a proof than merely to check the proof.

There is a cost, of course, in the loss of completeness, *i.e.* whereas exhaustive search at the object-level will eventually prove any theorem, our planners may

Name	Theorem	Source
$ass+$	$x + (y + z) = (x + y) + z$	BM14
$com+$	$x + y = y + z$	BM13
$com+_2$	$x + (y + z) = y + (x + z)$	BM12
$dist$	$x \times (y + z) = (x \times y) + (x \times z)$	BM16
$ass\times$	$x \times (y \times z) = (x \times y) \times z$	BM20
$com\times$	$x \times y = y \times x$	BM18
$tailrev_2$	$app(rev(a), n :: nil) = rev(n :: a)$	KF51
$assapp$	$app(l, app(m, n)) = app(app(l, m), n)$	BM05
$lensum$	$len(app(x, y)) = plus(len(x), len(y))$	us
$tailrev$	$rev(app(a, n :: nil)) = n :: rev(a)$	KF51
$lenrev$	$len(x) = len(rev(x))$	BM56
$revrev$	$x = rev(rev(x))$	BM47
$comapp$	$len(app(x, y)) = len(app(y, x))$	BM77
$apprev$	$app(rev(l), rev(m)) = rev(app(m, l))$	BM09
$applast$	$n = last(app(x, n :: nil))$	KF432
$tailrev_3$	$rev(app(rev(a), n :: nil)) = n :: a$	KF51

Key to Source Column
BMnn is theorem nn from appendix A of [Boyer & Moore 79].
KBnnn is example n.n.n from [Kanamori & Fujita 86].

Table 1: List of Theorems

Name	PT	OT	RT	PL	OL	RL	PS	OS	RS
ass+	1.0	73	73	3	160	53	7	$\sim 10^{8}$	$\sim 10^{7}$
ass+*	2.0	”	”	9	”	18	404	”	$\sim 10^{5}$
com+	2.2	93	42	7	~ 550	~ 80	25	$\sim 10^{12}$	$\sim 10^{10}$
com+*	4.3	”	”	18	”	~ 30	952	”	$\sim 10^{9}$
com+$_2$	2.1	109	52	5	~ 550	~ 110	39	$\sim 10^{15}$	$\sim 10^{13}$
com+$_2^{*}$	3.4	”	”	16	”	~ 30	14747	”	$\sim 10^{11}$
dist	17.1	405	24	12	~ 680	~ 60	$\sim 10^{6}$	$\sim 10^{32}$	$\sim 10^{26}$
ass$\times$	13.0	468	36	16	~ 1550	~ 100	$\sim 10^{5}$	$\sim 10^{35}$	$\sim 10^{30}$
com$\times$	11.3	372	33	17	~ 1850	~ 80	3078	$\sim 10^{32}$	$\sim 10^{28}$
tailrev$_2$	0.2	26	130	2	~ 25	~ 12	5	~ 800	~ 160
assapp	1.2	101	84	3	~ 300	~ 100	7	$\sim 10^{9}$	$\sim 10^{8}$
lensum	1.5	133	89	3	~ 200	~ 70	7	$\sim 10^{10}$	$\sim 10^{9}$
tailrev	1.8	212	118	4	~ 230	~ 60	17	$\sim 10^{10}$	$\sim 10^{9}$
lenrev	3.2	198	62	6	~ 550	~ 90	54	$\sim 10^{16}$	$\sim 10^{14}$
revrev	2.6	230	88	7	~ 900	~ 130	154	$\sim 10^{16}$	$\sim 10^{14}$
comapp	3.5	271	77	7	~ 400	~ 60	25	$\sim 10^{16}$	$\sim 10^{14}$
apprev	12.4	380	31	9	~ 520	~ 60	440	$\sim 10^{25}$	$\sim 10^{23}$
applast	-	-	-	-	-	-	-	-	-
tailrev$_3$	-	-	-	-	-	-	-	-	-

Key to Column Titles
First letter: P = Plan, O = Object-level, R = Ratio;
Second letter: T = Time, L = Length, S = Search Size;
e.g. RL is ratio of object-level proof length to plan length.
For more details see body of text.

A "-" sign indicates that either the planner or the tactic (as appropriate) failed on this problem.
A "*" indicates that the figures on this column are the results obtained without the *basic_plan*.
A "∼" sign indicates that this figure is an estimate.

Table 2: Results of Plan Formation and Execution

fail to find any plan for a theorem, or the all the plans that are found may fail to produce proofs. However, the high success rate of our current batch of tactics shows that this is not, yet, a practical problem. Completeness could, in any case, be regained by providing a low priority tactic which indulged in exhaustive search.

We have recorded two representative examples of theorems that our system cannot prove: *applast* and $tailrev_3$. *applast* is representative of a class of theorems which cannot be proved because Oyster cannot yet handle partial functions. In this case *last* is a partial function, being undefined on the empty list. Nuprl has recently been extended to handle partial functions, so it should not be too difficult to extend Oyster in the same way. With this extension, hand simulation suggests that our planner and tactics will succeed in planning and proving this and a number of similar theorems. $tailrev_3$ is representative of a more interesting class of theorems which involve an extension of our current set of tactics and methods, *e.g.* to include the ability to generalise sequents.

Our work is currently in the early stages. We have designed and implemented a few simple heuristics and tested them on some of the simpler examples from the literature. We have implemented a few simple planners for putting together these tactics. The methods and tactics proposed in [Bundy 88] required very little modification to prove the theorems listed in 1. By improving and extending our set of tactics and methods, over the next few months, we expect to be able to increase, significantly, the number of theorems that Oyster can prove.

5 Comparisons with Related Work

In this section we discuss the relationship of our work to that of other researchers building inductive theorem provers, others using tactics and others using meta-level inference.

As mentioned in §1, the state of the art in inductive theorem proving is still BMTP. We have yet to incorporate all the heuristics from BMTP into our tactics or to test them on the full range of theorems in [Boyer & Moore 79]. However, even on the simple examples we have tried so far we have found one improvement over BMTP; it can only prove $com\times$ if the lemma $u \times s(v) = u + u \times v$[3] has previously been proved. A combination of the fixed order of BMTP's heuristics and its inability to backtrack means that it misses the opportunity to propose and prove the lemma at the right moment and then it gets stuck down the wrong branch of the search space. The more flexible application of our tactics enables them to set up the key lemma they require[4] as a subgoal, and prove it, during the proof of $com\times$. Hence they do not require it to be pre-proved. In addition,

[3] A commuted version of the step case of the recursive definition of $\times$

[4] Which is a slight variant of the one required by BMTP

our experience of partially specifying and reasoning with inductive proof tactics has given us an insight into how the BMTP heuristics cooperate in the search for a proof and suggested ways of extending and improving them (see §6).

Tactics were first introduced to theorem proving in the LCF program verification system, [Gordon *et al* 79]. Their major use in LCF and Nuprl has been to automate small scale 'simplification' processes and to act as a recording mechanism for proof steps discovered by a human during an interactive session. We are unusual in using tactics to implement general-purpose whole-proof strategies, although there has been some work on the implementation of decision algorithms. We are unique in using plan formation to construct a purpose-built tactic for a theorem, although [Knoblock & Constable 86] discusses the (meta-)use of Nuprl to construct a tautology checking tactic from its specification.

Meta-level inference has been widely used in AI and logic programming to guide inference (see, for instance, [Gallaire & Lasserre 82]). However, most uses of meta-level inference have been to provide local control, *e.g.* to choose which subgoal to try to solve next or to choose which rule to solve it with. It has also been used for a coarse global control, *e.g.* to swap sets of rules in or out. We are unusual in using it to construct proof plans, *i.e.* outlines of the whole inference process. The only other use of proof plans we are aware of is earlier work in our own group, *e.g.* [Silver 85] and [Bundy & Sterling 88], on which this work builds, and the use of abstraction to build proof plans, *e.g.* [Sacerdoti 74]. Abstraction, in contrast to meta-level inference, works with a degenerate version of the object-level space in which some essential detail is thrown away. Because abstract plans are strongly tied to the object-level space, they are limited in their expressive power.

6 Limitations and Future Work

As mentioned in §5 we have not yet implemented all the heuristics from BMTP as tactics. In particular, we are still limited in the range of inductive rules of inference and recursive well-orderings and data-structures that we can handle.

In order to choose an appropriate form of induction, BMTP analyses the forms of recursion in the theorem to be proved. We call this process *recursion analysis.* We have yet to incorporate the full sophistication of this process into our proof plans, but we can see how to extend the preconditions of *basic_plan*, in a natural way, so that recursion analysis occurs as a side effect of plan formation. Indeed, we can see how to improve recursion analysis so that the form of induction used is not similar to any of the forms of recursion used in the statement of the theorem. We hope that this will, for instance, enable us to prove the standard form of the prime factorization theorem using the standard prime/composite form of induction, even though no prime/composite form of recursion appears in the theorem statement. This is beyond the BMTP in its current form.

At present there is a certain amount of redundancy in the work done by methods and tactics. For instance, comparison of the tactic and method for *unfold*, figures 2 and 4, respectively, shows that both calculate the step-equation and the result of the rewriting. We intend to reduce this redundancy by passing more information from the methods to the tactics via the tactic's arguments.

It is possible to calculate the output of our current simple tactics from the output and effects slots of their methods. As we build more sophisticated tactics we do not expect this to continue. The output pattern and the effects meta-formulae will only partially specify a tactic's output. It will then be necessary to satisfy the preconditions of subsequent methods not by evaluating them on the current sequent, but by a process of bridging inference from the effects of previous methods. This is a more expensive and open-ended process and needs careful control. Research into this extension continues.

Note that if *basic_plan* is not available as a tactic then the planner is able to reconstruct it by combining its sub-tactics (*cf.* figure 5). It would be nice to build a learning system that could remember such plans for future use. However, it would be necessary to weed out *ad hoc* plans that are not of general utility. Related work on learning plans from example proofs is being conducted within our group, [Desimone 87].

Our ideas on proof plans have been tested in the domain of inductive theorem proving because it is a challenging one in which there is a rich provision of heuristics. We have also done some earlier work in the domain of algebraic equation solving, [Silver 85]. We hope that proof plans will also be applicable in other domains. We have plans to explore their use in other areas of mathematics and in knowledge-based systems.

7 Conclusion

In this paper we have described empirical work to test the technique of proof plans, originally proposed in [Bundy 88], in the domain of inductive theorem proving. We have built a series of tactics for the proof checker, Oyster, partially specified these tactics using methods, and built a series of planners to construct proof plans from these methods. This system has proved a number of theorems drawn from the literature. The initial results are very encouraging; the planning search space is considerably smaller than the object-level one and plan steps are considerably cheaper than object-level steps. Our system has a high success rate on the simple theorems we have fed it. The rational reconstruction of the BMTP heuristics which has resulted from our expressing them in the form of tactics and methods has suggested a number of interesting extensions. Hand simulation of these suggests that we can build a theorem prover which will extend the state of the art.

Much work remains to be done in testing the technique of proof plans in this domain and in others, but preliminary results suggest that it will prove a powerful technique for overcoming the combinatorial explosion in automatic inference.

References

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

[Bundy & Sterling 88] A. Bundy and L.S. Sterling. Meta-level inference: two applications. *Journal of Automated Reasoning*, 4(1):15–27, 1988. Also available from Edinburgh, Dept. of AI, as Research Paper No. 273.

[Bundy 88] A. Bundy. *The Use of Explicit Plans to Guide Inductive Proofs.* Research Paper 349, Dept. of Artificial Intelligence, Edinburgh, 1988. Short version published in the proceedings of CADE-9. Long version submitted to JAR.

[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.

[Desimone 87] R.V. Desimone. Explanation-based learning of proof plans. In Y. Kodratoff, editor, *Machine and Human Learning*, Ellis Horwood, 1987. Also available as DAI Research Paper 304. Previous version in proceedings of EWSL-86.

[Gallaire & Lasserre 82] H. Gallaire and C. Lasserre. Metalevel control for logic programs. In *Logic Programming*, pages 173–185, Academic Press, 1982.

[Gordon *et al* 79] 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.

[Horn 88] 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.

[Kanamori & Fujita 86] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of Prolog programs. In *8th Conference on Automated Deduction*, pages 281–299, Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

[Knoblock & Constable 86] T. B. Knoblock and R.L. Constable. Formalized metareasoning in type theory. In *Proceedings of LICS*, pages 237–248, IEEE, 1986.

[Martin-Lõf 79] 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.

[Sacerdoti 74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Silver 85] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence.* North Holland, 1985. Revised version of the author's PhD thesis, DAI 1984.

3 More Things To Try

Here are some more proofs for you to exercise on. There are proofs of them in the benchmarks directory, but they will probably seem a bit obscure. Try them from scratch, and don't expect your solution to look anything like the benchmark one.

1. `[]>>a:u(1)->(a\a->void)->((a->void)->void)->a`

   aka $a \vee \neg a \to \neg\neg a \to a$

2. `[]>>a:u(1)->p:(a->u(1))->(((x:a->p of x)->void)->void)->x:a->(p of x->void)->void`

   $(\neg\neg\forall x p(x) \to \forall x \neg\neg p(x))$

3. `[]>>a:u(1)->p:(a->u(1))->(x:a#p of x->void)->(x:a->p of x)->void`

   $(\exists x \neg p(x) \to \neg\forall p(x))$

4. `[]>>a:u(1)->p:(a->u(1))->((x:a#p of x->void)->void)->x:a->(p of x->void)->void`

   $(\neg\neg\exists x p(x) \to \forall x \neg\neg p(x))$

5. `[]>>a:u(1)->p:(a->u(1))->((x:a#p of x)->void)->x:a->p of x->void`

   $(\neg\exists x p(x) \to \forall x \neg p(x))$

6. `[plus(x,y)<==>p_ind(x,y,[~,v,s(v)])]`

   `>>x:pnat->y:pnat->z:pnat->plus(x,plus(y,z))=plus(plus(x,y),z)in pnat`

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

   This is a bit harder than the others. It involves:

   - A definition presented as a hypothesis.
   - Understanding the recursive definition functional *p_ind* – it allows you to refer to recursion over the natural numbers. It works much like NuPRL's *ind* and *list_ind*. The first argument is the recursion argument, the second is the function's value if the recursion argument is 0, and the third describes how to compute its value if it is of the form $s(x')$ This third argument is a triple, of which the first two elements are x' and the value of the function being defined applied to x'. The third element of the triple is then the value of the function in terms of the first two elements of the triple. In this case, the first element of the triple is not required and is left as ~.