

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 synthesis-time type checking of 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 the **BMTP** would apply these tactics, successively, 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 \vdash *true*s as the final ones. 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 a **BMTP**-style theorem prover would fail rather than backtrack, *i.e.* 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 it 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, collectively called **CLAM**, which each 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 by giving some preconditions for its attempted application and some effects of its successful application. We call this 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

2.1 Example Inductive Proofs

Figure 1 is a simple illustrative example of the kind of proof generated by the **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 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 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.

$\forall u : \text{pnat}. \{0 + u = u\}$		↑
$\forall v : \text{pnat}, \forall w : \text{pnat}. \{s(v) + w = s(v + w)\}$		
$x : \text{pnat}$		
$y : \text{pnat}$		
$z : \text{pnat}$		
$\vdash \overline{x + (y + z) = (x + y) + z}$		
<i>induction</i>		
$\vdash \overline{0 + (y + z) = (0 + y) + z}$	$\overline{x' : \text{pnat}}$	↑ <i>i</i>
$2 \times \text{base}$	$\overline{x' + (y + z) = (x' + y) + z}$	↑ <i>n</i>
$\vdash \overline{y + z = \overline{y} + z}$	$\vdash \overline{s(x') + (y + z) = (s(x') + y) + z}$	<i>r d</i>
<i>sym_eval</i>	$2 \times \text{wave (1st wave)}$	<i>i</i>
$\vdash \overline{\text{true}}$	$\vdash \overline{s(x' + (y + z)) = \overline{s(x' + y) + z}$	<i>p s</i>
	<i>wave</i>	<i>p t</i>
	$\vdash s(x' + (y + z)) = \overline{s(x' + y) + z}$	<i>l r</i>
	<i>fertilize</i>	<i>e a</i>
	$\vdash \overline{s(x' + (y + z)) = \overline{s(x' + (y + z))}$	<i>o t</i>
	<i>sym_eval</i>	<i>u</i>
	$\vdash \overline{\text{true}}$	<i>t</i>
		↓
		↓

Figure 1: Outline Proof of the Associativity of +

The proof in figure 1 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 *base* and *wave* tactics then rewrite the base and step case, respectively, using the base and step equations of the recursive definition of +. The two applications of *base* rewrite the base case to an equation between two identical expressions, which the *sym_eval* tactic reduces to $\vdash \text{true}$. The three applications of *wave* 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. Following [Aubin 75], we call this process *rippling-out*. The two arguments of the successor functions are then identical to the two arguments of = in the induction hypothesis. The *fertilize* tactic then replaces the right hand of these two arguments by the left hand one in the induction conclusion. The two arguments of the successor functions are now identical and the *sym_eval* tactic reduces the sequent to $\vdash \text{true}$. 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 sub-tactics *induction*, *base*, *wave* and *fertilize* in the order suggested by the above proof.

As further evidence of the generality of this proof structure consider the proof of the *even+* theorem.

$$(\text{even}(x) \wedge \text{even}(y)) \rightarrow \text{even}(x + y)$$

which is given in figure 2. This uses the same tactics in a very similar combination, but uses a

where the *wave term* is indicated by the underbrace, the *wave function* is indicated by the overbrace and U is the *wave argument*. The idea is that, initially, U is instantiated to the induction variable and $S(U)$ to the term that is substituted for the induction variable in the induction conclusion, e.g. $s(x)$ in figure 1 and $s(s(x))$ in figure 2. Applying a wave rule ripples this term out past the first of the nested functions that contain it. In the process, S is transformed to T . The next application of a wave rule ripples T out past the function that immediately dominates it and so on.

Examples of wave rules that are step cases of inductive definitions are:

$$\begin{aligned} \underbrace{s(u)} + v &\Rightarrow s(\overbrace{u+v}) \\ \text{even}(\underbrace{s(s(u))}) &\Rightarrow \overbrace{\text{even}(u)} \\ \text{app}(\underbrace{n :: u, l}) &\Rightarrow n :: \overbrace{\text{app}(u, l)} \end{aligned}$$

Examples of wave rules that are not step cases of inductive definitions are:

$$\begin{aligned} \text{even}(\underbrace{u+v}) &\Rightarrow \overbrace{\text{even}(u)} \wedge \overbrace{\text{even}(v)} \\ \text{product}(\underbrace{\text{app}(ul, vl)}) &\Rightarrow \overbrace{\text{product}(ul)} \times \overbrace{\text{product}(vl)} \\ \underbrace{u_1 \times u_2} = \underbrace{v_1 \times v_2} &\Rightarrow \overbrace{u_1 = v_1} \wedge \overbrace{u_2 = v_2} \end{aligned}$$

In our meta-language, the existence of a wave rule for a particular function is represented by an assertion of the form:

$$\text{wave_rule}(F(U), N, S(U), \text{Name})$$

where $F(U)$ is the wave function, N is the number of the wave argument, $S(U)$ is the wave term, and Name is a pointer to the rule. Examples of the use of this assertion can be seen in the preconditions of the *ind_strat* method in §3.

2.3 An Example Tactic

Each of our 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. A complete list of the tactics we use is given in appendix B. The Prolog procedure, *ind_strat/2*, for the *ind_strat* tactic, is given in figure 3 and discussed below.

The two arguments of *ind_strat/2*: the induction schema, **Schema**, and the induction variable and type, **Var:T**, (0). Currently *ind_strat* can only deal with induction schemata that have a single step case, but may have any number of base cases. This limitation can be easily overcome. Literals (1) and (2) find out how many base cases there are. Literals (3) and (4) construct a tactic for dealing with these base cases and the step case. These consist of repeated applications of **base** to the base case (3) and repeated applications of **wave** to the step case (4). The **iterator** tactical applies a tactic repeatedly. It succeeds if the first application succeeds. Finally, **fertilize** is tried on the step case (4). The **try** tactical applies a tactic and succeeds whether or not this application succeeds. The **then** tactical links these sub-tactics together. Literal (5) applies induction to the conjecture and then applies the constructed base and step tactics.

Note that the tactic will still succeed and return a result even if the rippling out is not completed or the **fertilize** is not successful. Only the **induction** and the first applications of **wave** are necessary for success.

```

ind_strat(Schema,Var:T) :- (0)
  clause(schema(Schema,Var:T,_,BaseCases,_) ,_), (1)
  length(BaseCases,NrOfBaseCases) (2)
  makelist(NrOfBaseCases,iterator(base,_) ,BaseTactics), (3)
  append(BaseTactics, (4)
    [iterator(wave,_) then try fertilize(IndHyp)],
    BaseAndStepTactics),
  apply(do_induction(Schema,Var:T,IndHyp) (5)
    then BaseAndStepTactics).

```

Figure 3: Prolog Code for the `ind_strat` Tactic

A selected list of theorems to which these tactics have been applied is given in table 1. The definitions of the functions used in these theorems are given in appendix C. 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.

3 Using Planning for Flexible Application

We have had some success in proving theorems by repeated application of the *ind_strat* and *sym_eval* tactics. This success confirms the hypothesis proposed in [Bundy 88] that the proof structure captured in *ind_strat* 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 *base* 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 the 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 6 and 7. The theorems are then proved by executing the super-tactics defined by these plans. The system responsible for forming these proof plans is called CLAM.

A method is represented as an assertion of the Prolog procedure *method/6* in the format given in figure 4. 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 fourth argument, (4), is the *output formulae*, a list of meta-level patterns which any formulae output by the tactic will match. The fifth argument, (5), 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.

There are methods for each of the tactics listed in appendix B. The method for the *ind_strat* tactic is given, as an example, in figure 5. The declarative meaning of the predicates in this and the other methods can be found in appendix A. The first and last arguments of *method/6*, (1) and (6), are identical because of our decision to overload the tactic names and use them for the method names. In this case both names are `ind_strat` and both have the same two arguments: the induction schema, `Schema`, and the induction variable with its type, `Var:T`. The input to the *ind_strat* tactic, (2), is a sequent of the form $H \mid - G$.

<i>method</i> (<i>name</i> (... <i>Args</i> ...),	(1)
<i>Input formula</i> ,	(2)
<i>Preconditions</i> ,	(3)
<i>Output formulae</i> ,	(4)
<i>Effects</i> ,	(5)
<i>tactic</i> (... <i>Args</i> ...)	(6)
)	

Figure 4: The Format of Methods

The preconditions, (3), are used to suggest values for **Schema** and **Var** given $H \vdash G$. **Var:T** must originally have been universally quantified and, hence, must now be a hypothesis, (3). The remainder of the preconditions concern the existence of wave rules whose wave function matches a term containing **Var**. These are called *wave occurrences* of **Var**. There must be at least one wave occurrence of **Var**, (3a). For each wave occurrence **Schema** must subsume the wave term of its wave rule, (3b). Optionally, all occurrences of **Var** must be wave occurrences, (3c).

These preconditions look ahead to the ripple-out process and try pick an induction schema and induction variables that will ensure that it succeeds. Unfortunately, it is impossible to guarantee success in all cases: a key wave rule might be missing and need to be proved as a lemma at run time. Moreover, it is extremely messy to try to predict the whole rippling-out process. So our actual preconditions are a compromise. We only look ahead one *wave* application for each occurrence, (3b). If there is no option, we are prepared to accept induction variables with occurrences for which there is no wave rule, (3c). If these preconditions are satisfied then the tactic is guaranteed to succeed. Ideally, we would have clever interpreters for method and tactic application which could cope with partial success of strong preconditions, effects and tactics. However, as a short term expedient, we have designed the ‘cleverness’ into the methods and tactics by the use of connectives like **andpossibly**, **iterator** and **try**. This has the unfortunate side effects of making them hard to read and more brittle than desirable. For an extended discussion of the preconditions of the *ind_strat* method, see [Bundy *et al* 88].

The output of the tactic will be a list of sequents corresponding to the remains of the base and step cases, (4). The effects of the tactic are calculated by considering the manipulations of the various subtactics, (5). The procedure which does this is similar to the tactic itself (figure 3) except that it is not necessary to do the manipulations by calling Oyster rules of inference and, in particular, it is not necessary to check the legality of each step (*e.g.* well-foundedness checks are not needed). This makes it much cheaper, and potentially fallible, although not in this case.

Finding proof plans presents an unusual plan formation problem. Most AI planners work backwards from the final goal¹ to the initial state. Unfortunately, the final goal of all our proofs is a list of \vdash *true*s, 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.

Each planner starts by inspecting the current sequent (initially the conjecture to be proved), and searches for applicable methods. A method is applicable whenever its input slot matches the current sequent, and its preconditions succeed. After choosing one of these applicable methods, a planner computes a list of output sequents by evaluating the effects slot of the selected method, and checking that the result matches the output slot of the method. Each member of this list of output sequents will then serve as the input sequent for the next cycle of the planning process, until all the output sequents have the form \vdash *true*.

¹Note that goals in planning are not the same thing as goals in sequents.

```

method(ind_strat(Schema,Var:T), (1)
  H |- G, (2)
  [(hyp(Var:T,H), (3)
    exists {WavePos}: wave_occ(G,Var,WavePos,_,_,_), (3a)
    exists {Schema\schemata(Var:T,Schema)}: (3b)
      (forall {(F,N)\wave_occ(G,Var,[N|_],F,_,_)}:
        (wave_rule(F,N,S,_), subsumes(Schema,S)))
    )
  andpossibly
  forall {(F,N)\occ(G,Var,[N|_],F)}: (3c)
    wave_rule(F,N,_,_)
  ],
  [All|Seqs], (4)
  [schema(Schema, Var:T, H |- G, BaseSeqs, StepSeq), (5)
  maplist(BaseSeqs, BaseSeq:=>BaseSeq1,
    applicable(BaseSeq, try iterator(base,_,_), [BaseSeq1]),
    BaseSeq1s),
  applicable(StepSeq, iterator(wave,_,_), [StepSeq1]),
  applicable(StepSeq1, try fertilize(.,_,_), [StepSeq2]),
  append(BaseSeq1s, [StepSeq2], [All|Seqs])
  ],
  ind_strat(Schema,Var:T) (6)
).

```

Figure 5: The Method for the *ind_strat* Tactic

During this planning cycle, a number of choice points occur: more than one method may be applicable to the input sequent, and one method may apply in more than one way (i.e. its preconditions may be satisfied in more than one way). The planners we have built differ only in the strategy they employ at these choice-points. Some make cheap but rather uninformed choices, some try to make a more informed choice; some of them make sure that all choices get equal treatment, others favour one choice over others, *etc.*

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 plan². Our *best-first* planner is only slightly slower than the depth-first planner and, in practice, usually terminates with plans of reasonable length. Its simple heuristics consist of preferring unbranching methods (such as *wave*) over branching methods (such as *induction*), and preferring methods which cannot give rise to infinite loops (such as *base*). Also, large scale methods which consist of the application of a number of smaller methods (such as *ind_strat*) are tried before the smaller methods are tried individually.

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 \vdash *true*s. At each cycle it finds a method that is applicable

² Although the theoretical time complexity of breadth-first search is better than that of iterative-deepening by a small constant (since breadth-first search does not have to compute any partial results), our iterative deepening planner is actually faster than the breadth-first planner. This is because of two factors. Firstly, iterative deepening can use the Prolog search strategy directly, whereas breadth-first search has to support its own search strategy in a very memory intensive way, and secondly our iterative deepening planner is able to exploit the linearity property of the planning space.

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.

For instance, the initial state when proving the *ass+* example is the sequent:

$$\dots, x:pnat, y:pnat, z:pnat \vdash x + (y + z) = (x + y) + z \quad (1)$$

The first method whose preconditions are applicable is *ind_strat*. During the satisfaction of its preconditions (see figure 5) against (1) the arguments **Schema** and **Var:T** are instantiated. Firstly, (2), $\mathbb{H} \vdash \mathbb{G}$ is matched against the initial state, (1). x, y and z are all universal variables declared in the hypothesis, (3). Both x and y have wave occurrences, (3a). Both the wave occurrences of x are subsumed by the schema $\mathbf{s}(x)$ and the single wave occurrence of y is subsumed by the schema $\mathbf{s}(y)$, (3b). Only for x is it true that all occurrences are wave occurrences, (3c). Therefore, **Var:T** is instantiated to $\mathbf{x:pnat}$ and **Schema** is instantiated to $\mathbf{s}(x)$.

Applying this induction schema to (1) produces one base and one step case, as in figure 1. The effects of the method, (5), then predict the effect of applying the *ind_strat* tactic to (1). After this application the remains of the base and step cases are predicted to be as in figure 1, namely:

$$\begin{aligned} \dots \vdash y + z = y + z \\ \dots \vdash s(x' + (y + z)) = s(x' + (y + z)) \end{aligned}$$

and these are returned in a list of output sequents, (4). *sym_eval* is the first method applicable to each of these subgoals. It reduces each of them to $\vdash true$ using the reflexive rule of equality. The resulting plan is displayed on the left hand side of figure 7. If the *ind_strat* method is not available, then the planner can put together its various sub-methods to make a special case of *ind_strat*. This plan is displayed in figure 6.

When the tactic $\mathbf{ind_strat}(\mathbf{s}(x), \mathbf{x:pnat})$ is executed it generates the top part of the proof outlined in figure 1, as required. In Oyster this proof consists of 139 rule of inference applications! This 139:1 ratio indicates the gearing that we get from planning the proof. Further evidence for this can be found in table 2. 108 of these applications are concerned with proving well-typedness, but even if these are ignored the remaining 31:1 ratio still indicates a significant gearing.

The initial state when proving the *com+* example is the sequent:

$$\dots, x:pnat, y:pnat \vdash x + y = y + x$$

When the *ind_strat* method is applied to this state its preconditions succeed with induction variable $\mathbf{x:pnat}$ and induction schema $\mathbf{s}(x)$. However, both universal variables have one wave occurrence and one non-wave occurrence, so both fail the optional part (3c) of the preconditions, and both are equally applicable. \mathbf{x} is chosen arbitrarily. Because the induction suggestion is flawed in this way the rippling-out process breaks down, although it succeeds enough to allow *fertilize* to succeed. However, since *induction*, one *base* and one *wave* are successfully applied the method application is deemed to have succeeded. The output sequents returned are:

$$\begin{aligned} \dots \vdash y = y + 0 \\ \dots \vdash s(y + x') = y + s(x') \end{aligned}$$

Recursive applications of $\mathbf{ind_strat}(s(y), y:pnat)$ to each case followed by four applications of *sym_eval* are required to complete the proof. Note that a wave rule of the form:

$$y + s(x') \Rightarrow s(y + x')$$

would have been enough to unstick the rippling-out, and this wave rule is effectively proved ‘in-line’ by the recursive application of $\mathbf{ind_strat}(s(y), y:pnat)$ to the step case. The right hand side of figure 7 shows the complete plan. This example illustrates the way in which *ind_strat* can be nested in a plan.

```

induction(s(x), x : pnat) then
  [ base ([1, 1, 1]) then
    base ([1, 1, 2, 1]) then
      sym_eval ,
    wave([1, 1, 1]) then
      wave([1, 1, 2, 1]) then
        wave([1, 2, 1]) then
          fertilize ([1], v3) then
            sym_eval
        ]
  ]

```

Figure 6: The Proof Plan Generated for *ass+*

```

ind_strat(s(x), x : pnat) then      ind_strat(s(x), x : pnat) then
  [ sym_eval,                       [ ind_strat(s(y), y : pnat) then
    sym_eval                          [ sym_eval,
  ]                                    sym_eval
                                     ],
                                     ind_strat(s(y), y : pnat) then
                                     [ sym_eval,
                                     sym_eval
                                     ]
                                     ]
]

```

Figure 7: The Plans for *ass+* and *com+* using the *ind_strat* 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 (or iterative-deepening planner in the two cases indicated by the “†” sign). 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 because the depth-first planner sometime gets trapped down infinite branches. With the iterative-deepening planner, times are usually slightly longer than for the best-first planner. In some cases, iterative-deepening times are considerably longer, and resource limitations causes it to fail where the best-first planner succeeds. This is why estimates of PS are given for two examples. In other cases, the incompleteness of the best-first planner causes it to fail where the iterative-deepener succeeds. This is why iterative-deepening times are given for two examples. With the breadth-first planner, times are several orders of magnitude longer and many planning attempts had to be abandoned due to resource limitations. Most figures are calculated

with the *ind_strat* method available, but for some of the simpler theorems we also give figures without *ind_strat*. These rows are marked by a * against the theorem name. Note that it takes longer to find a plan when the *ind_strat* method is not available, although our planners can find plans *not* containing the *ind_strat* for all those theorems for which they can find plans containing the *ind_strat*.

- OT — is the time in cpu seconds to execute the plan by running its associated Oyster tactic. This tactic calls rules of inference of our 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 plan found by the best-first planner, *i.e.* the number of tactics in the plan. Note that plans are longer when the *ind_strat* tactic is not available. This is because one *ind_strat* step unpacks into several *induction*, *wave*, *etc.* steps. The planning process finds these shorter, *ind_strat* plans before the longer ones.
- 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. The figures in parentheses indicate the number of applications of rules not concerned merely with type information.
- 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 size of the planning search space that is searched by the iterative-deepening planner before it finds a plan. Note that there is much less search when the *ind_strat* is available. 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 size of the object-level search space that would be searched by the iterative-deepening planner before it would find a proof. 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. This is why estimates 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. This means that the figures in the PT and PS columns are not different measures of the same process and, hence, are not in proportion. Compare, for instance, the figures for *com+₂* and *com+₂**. The figures for PS give the better estimate of the size of the planning search space. The figures for PT show how well the best first heuristics cope with this search space.

These results are very encouraging. The much smaller search space required for planning as opposed to theorem proving (see column RS) shows a considerable ability to defeat the

Name	Theorem	Source
<i>ass+</i>	$x + (y + z) = (x + y) + z$	BM14
<i>com+</i>	$x + y = y + z$	BM13
<i>com+₂</i>	$x + (y + z) = y + (x + z)$	BM12
<i>dist</i>	$x \times (y + z) = (x \times y) + (x \times z)$	BM16
<i>ass×</i>	$x \times (y \times z) = (x \times y) \times z$	BM20
<i>com×</i>	$x \times y = y \times x$	BM18
<i>even+</i>	$(\text{even}(x) \wedge \text{even}(y)) \rightarrow \text{even}(x + y)$	us
<i>primes</i>	$x \neq 0 \rightarrow \exists l: \text{list}(\text{primes}).\text{prod}(l) = x$	us
<i>tailrev₂</i>	$\text{app}(\text{rev}(a), n :: \text{nil}) = \text{rev}(n :: a)$	KF51
<i>assapp</i>	$\text{app}(l, \text{app}(m, n)) = \text{app}(\text{app}(l, m), n)$	BM05
<i>lensum</i>	$\text{len}(\text{app}(x, y)) = \text{len}(x) + \text{len}(y)$	us
<i>tailrev</i>	$\text{rev}(\text{app}(a, n :: \text{nil})) = n :: \text{rev}(a)$	KF51
<i>lenrev</i>	$\text{len}(x) = \text{len}(\text{rev}(x))$	BM56
<i>revrev</i>	$x = \text{rev}(\text{rev}(x))$	BM47
<i>comapp</i>	$\text{len}(\text{app}(x, y)) = \text{len}(\text{app}(y, x))$	BM77
<i>apprev</i>	$\text{app}(\text{rev}(l), \text{rev}(m)) = \text{rev}(\text{app}(m, l))$	BM09
<i>applast</i>	$n = \text{last}(\text{app}(x, n :: \text{nil}))$	KF432
<i>tailrev₃</i>	$\text{rev}(\text{app}(\text{rev}(a), n :: \text{nil})) = n :: a$	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
<i>ass+</i>	1.0	73	73	3	160(34)	53	7†	$\sim 10^8$	$\sim 10^7$
<i>ass+*</i>	2.0	"	37	9	"	18	404†	"	$\sim 10^5$
<i>com+</i>	2.2	93	42	7	182(60)	26	25†	$\sim 10^{12}$	$\sim 10^{10}$
<i>com+*</i>	4.3	"	22	18	"	10	952†	"	$\sim 10^9$
<i>com+₂</i>	2.1	109	52	5	225(50)	45	39†	$\sim 10^{15}$	$\sim 10^{13}$
<i>com+₂*</i>	3.4	"	32	16	"	14	14747†	"	$\sim 10^{11}$
<i>dist</i>	17.1	405	24	12	811(140)	67	$\sim 10^6$	$\sim 10^{32}$	$\sim 10^{26}$
<i>ass×</i>	13.0	468	36	16	882(145)	55	$\sim 10^5$	$\sim 10^{35}$	$\sim 10^{30}$
<i>com×</i>	11.3	372	33	17	665(144)	39	3078†	$\sim 10^{32}$	$\sim 10^{28}$
<i>even + *</i>	24.8	55	2.2	6	244(90)	41	28†	$\sim 10^{18}$	$\sim 10^{16}$
<i>primes</i>	176†	358	2.0	10	553(88)	55	78	$\sim 10^{25}$	$\sim 10^{23}$
<i>tailrev₂</i>	0.2	26	130	2	68(7)	34	5†	~ 800	~ 160
<i>assapp</i>	1.2	101	84	3	209(32)	69	7†	$\sim 10^9$	$\sim 10^8$
<i>lensum</i>	1.5	133	89	3	245(34)	81	7†	$\sim 10^{10}$	$\sim 10^9$
<i>tailrev</i>	1.8	212	118	4	433(48)	108	17†	$\sim 10^{10}$	$\sim 10^9$
<i>lenrev</i>	3.2	198	62	6	333(58)	55	54†	$\sim 10^{16}$	$\sim 10^{14}$
<i>revrev</i>	2.6	230	88	7	578(64)	82	154†	$\sim 10^{16}$	$\sim 10^{14}$
<i>comapp</i>	3.5	271	77	7	326(50)	46	25†	$\sim 10^{16}$	$\sim 10^{14}$
<i>apprev</i>	12.4	380	31	9	727(103)	80	440†	$\sim 10^{25}$	$\sim 10^{23}$
<i>applast</i>	-	-	-	-	-	-	-	-	-
<i>tailrev₃</i>	-	-	-	-	-	-	-	-	-

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 the ratio of the object-level proof length to the plan length.

For more details see body of text.

A “-” sign indicates that the planner failed on this problem.

A “*” indicates that the figures on this column are the results obtained without the *ind_strat*.

A “†” sign indicates that the iterative-deepening planner was used.

A “ \sim ” sign indicates that this figure is an estimate.

Figures in parentheses are the number of non-type oriented proof steps.

Table 2: Results of Plan Formation and Execution

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 RT shows that it is much *cheaper* to search in the planning space than to execute the plan at the object level, even though the latter involves no search. 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.

One could object that the huge object-level space consists largely of typing rules which are easily controlled by standard Nuprl/Oyster sub-tactics, *e.g.* *wfftacs*, so that the savings gained from planning are more apparent than real. However, the figures in brackets in the OL column indicate that, even without these typing rules, the object-level proofs and, hence, the object-level spaces are considerably bigger than the planning spaces. So planning brings very significant efficiencies, even if the typing rules are factored out. One can draw similar conclusions by comparing the figures with and without the use of the *ind_strat* for each of the theorems for which these were given. The ‘without’ figures serve as a sort of object-level to the ‘with’ figures meta-level. It can be seen that the introduction of the additional layer of planning provided by the *ind_strat* gives considerable decreases in planning time, proof length and amount of search.

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 fail to find any plan for a theorem. 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 *tailrev₃*. *applast* is representative of a class of theorems which cannot be proved because Oyster cannot yet handle partial functions. In this case *last* is naturally represented as a partial function, being undefined on the empty list. *last* can be defined as a total function by defining it to take some arbitrary value on the empty list, but then the form of recursion is unusual and our tactics cannot yet handle it. Hand simulation suggests that, with some simple amendments, our planner and tactics will succeed in planning and proving this and a number of similar theorems. *tailrev₃* 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 expression.

It should be noted that the best-first and depth-first planners fail fairly quickly on typical false conjectures. For example, the depth-first planner fails in under ten seconds on the conjecture $x \times y = x + s(0)$.

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. By improving and extending our set of tactics and methods 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 doing related work. We include work on (a) building inductive theorem provers, (b) using tactics and (c) using meta-level inference.

However, we start by comparing the CLAM and Oyster systems described here with the original proposal for proof plans in [Bundy 88]. The methods and tactics we proposed in [Bundy 88] required very little modification to prove the theorems listed in table 1. Since that paper was written there has been a steady, evolutionary process of improvement and extension, as our understanding of the induction strategy developed. We also rationalised their names, *e.g.* *ind_strat* used to be called *basic_plan*.

Apart from this, the main changes were to the nature of the effects. The effects in [Bundy 88] were weak meta-level assertions about the relation between the input and the output of the tactic, which permitted only a partial prediction about the form of the output. The CLAM effects are strong meta-level assertions which permit a complete prediction of the output from the input. CLAM effects can be run as programs and mimic the behaviour of the tactics. We call them *pseudo-tactics*. Using pseudo-tactics simplified the design of the planners and made them efficient in operation. Pseudo-tactics do the same manipulations as tactics, but much more cheaply. This cheapness is one major factor in making it cheaper to pre-plan proofs than to search directly in the object-level search space. The other factor is the grouping of object-level steps into bigger planning steps.

Another difference from the methods proposed in [Bundy 88] is that the success of CLAM preconditions ensures the success of CLAM effects and hence of the Oyster tactics *i.e.* a CLAM method is a complete specification of its tactic. Thus CLAM proof plans never fail. However, this guarantee is bought at the price of building optional steps into the preconditions, effects and tactics using **andpossibly**, **xor**, *etc.*, which makes them fairly messy. We are considering reverting to the proposal in [Bundy 88] of making methods be partial specifications of tactics, where the preconditions do not imply the effects. However, this will require much more flexibility in the planning and the tactic application mechanisms.

As mentioned in §1, the state of the art in inductive theorem proving is still the BMTF. We have yet to incorporate all the heuristics from the BMTF 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 some improvements over the BMTF. For instance, BMTF can only prove *com*× if the lemma³ $u \times s(v) = u + u \times v$ has previously been proved. A combination of the fixed order of the BMTF'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 require⁴ as a subgoal, and prove it, during the proof of *com*×. Hence they do not require it to be pre-proved. In addition, our experience of partially specifying and reasoning with inductive proof tactics has given us an insight into how the BMTF heuristics cooperate in the search for a proof and suggested ways of extending and improving them (see §6).

In order to choose an appropriate form of induction, the BMTF 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 a simple form of recursion analysis occurs as a side effect of fitting the *ind_strat* to the theorem during plan formation. For more details see [Bundy *et al* 88]. This is sufficient, for example, to deal with the various induction possibilities in the *even+* example above.

Our version of recursion analysis improves on the BMTF version in various ways. For instance, the form of induction CLAM selects need not be similar to any of the forms of recursion used in the statement of the theorem. This extension has enabled us to prove the classic form of the prime factorization theorem (*primes* in table 1) using the classic prime/composite form of induction, even though no prime/composite form of recursion appears in the theorem statement. This is beyond the BMTF in its current form. For more details see [Bundy *et al* 88].

This theorem involves existential quantification, which again represents an extension to the BMTF. We deal with this by substituting a meta-variable for the existential variable when eliminating the existential quantifier during the planning process. Unification then instantiates this to an appropriate object-level term as the planning process progresses, *e.g.* during rippling-out. When the *ind_strat* tactic is applied this object-level term is substituted for the existential variable at the time of existential quantifier elimination.

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 'simpli-

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

⁴Which is a slight variant of the one required by the BMTF

fication’ processes and to act as a recording mechanism for proof steps discovered by a human during an interactive session. Nuprl, for example has a tactic which will exhaustively perform certain forms of symbolic evaluation, including the unfolding of definitions. This is sufficient to complete some proofs where only a single induction is required, after the correct induction has been applied; but in general more flexibility is required. For example, where the symbolic evaluation tactic is not strong enough to prove the goal on its own, what is required may be the exploitation of the recursive properties of *some* of the definitions present. Exhaustive unfolding and evaluation of the definitions in this situation will greatly obscure the structure of the resultant subgoal, and so make the subsequent proof search more difficult. Proof plans allow the appropriate flexible application of symbolic evaluation.

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 which can prove whole theorems and in implementing weak, general purpose, search strategies, like best-first search. 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.

The Isabelle theorem prover due to Paulson ([Paulson 88]) represents one extension of the LCF approach. It is a generic theorem prover incorporating unification (in this case higher-order) and supporting schematic assertion. Isabelle extends the given object-level logic by means of derived inference rules implemented as tactics; these derived rules have the same status as the primitive inference rules, counting as a single proof step, so hiding the structure of the proof in terms of the original logic. For us, the proofs themselves are objects of interest — we are not just interested in provability. We can, for example, optimise the program synthesised by a proof using operations that transform the structure of the proofs. Accordingly, our tactics construct a proof in terms of the basic inference rules. Whereas in Isabelle, inference rules are chained together by unification, we have a distinct and explicit planning level, which may use simple pattern matching, but where more strategic control can come into play. Unlike Isabelle’s tactics, proof plans are not limited to derived inference rules. The meta-language can describe classes of rules and lemmas and form tactics from arbitrary long sequences of these.

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 based on 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 the 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 type theory it is possible to build pseudo-tactics that are considerably more efficient to run than the tactics, but which make the same syntactic manipulations to the input formula. It will not be possible to write cheap pseudo-tactics in all mathematical theories. One solution to this problem is to ensure that output prediction remains cheap by only expecting method application to calculate a partial description of the output. For instance, it might only return a pattern, *i.e.* a formula containing meta-variables. The *existential* method already does this. Alternatively, the method application might return an abstraction of the output. Or, we might

revert to the proposal in [Bundy 88], and return a meta-level description of the 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. It also makes it much more likely that plans will be fallible, requiring re-planning after a custom-built tactic has failed to behave as predicted.

Note that if *ind_strat* is not available as a tactic then the planner is able to reconstruct it by combining its sub-tactics (*cf.* figure 6). 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 86].

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 intend 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 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 BMTF heuristics which has resulted from our expressing them in the form of tactics and methods has suggested a number of interesting extensions, some of which have been tested and which have extended 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.

A The Meanings of the Terms Used in the Meta-Logic

The declarative meanings of the meta-logical predicates and functions used in the methods are given below. These are implemented as Prolog procedures. Their input modes as Prolog procedures is indicated by the prefixes on the argument names. A “+” indicates an argument bound at execution time (an input). A “-” indicates an argument unbound at execution time (an output). A “?” indicates an argument which is sometimes bound and sometimes unbound at execution time.

<code>base_rule(?Func,?BaseEqs)</code>	BaseEqs is the list of base-equations for Func.
<code>base_rules(?Rules)</code>	Rules is the list of all base-equations.
<code>canonical_form(+Exp,+Rules,?New)</code>	Expression New is the canonical form of Exp using all elements of Rules as rewrite rules.
<code>exp_at(+Exp,?Pos,?SubExp)</code>	SubExp is the subexpression in Exp at position Pos
<code>groundp(+Term)</code>	Term is ground (does not contain meta-variables).
<code>hyp(?Hyp,?HypList)</code>	Hyp is a member of HypList.
<code>instantiate(+G1,?G2,?G2Vals)</code>	G2 is obtained by instantiating all universally quantified variables in G1 with the values G2Vals.
<code>metavar(?Var)</code>	Var is a meta-variable.
<code>nr_of_occ(?SubExp,+SupExp,?N)</code>	SubExp occurs exactly N times in SupExp.
<code>occ(+Term,?SubTerm,?Pos,?F)</code>	SubTerm occurs in Term at Pos, immediately surrounded by term F.
<code>recursive(?Term,?Num,?Schema)</code>	Term is a function that is recursive in its Numth argument according to recursion Schema.
<code>replace(+Pos,?Sub,+Old,?New)</code>	Expression New is the result of replacing the subexpression in Old at position Pos with Sub.
<code>replace_all(+S1,+S2,+Old,?New)</code>	Expression New is the result of replacing all occurrences of S1 with S2 in Old.
<code>rewrite(?Pos,+Rule,?Exp,?New)</code>	Expression New is the result of rewriting the subexpression in Exp at position Pos using equation Rule.
<code>schema(+S,+V,+Seq,?BSeqs,SSeq)</code>	BSeqs and SSeq are the list of base-sequents and the step-sequent produced by applying induction schema S to variable V in Seq.
<code>simplify_rules(?Rules)</code>	Rules is the set of all base-, step- and wave-equations.
<code>subsumes(+S1,+S2)</code>	Induction schema S1 subsumes induction schema S2.
<code>step_rule(?Func,?StepEqs)</code>	StepEqs is the step-equation for Func.
<code>step_rules(?Rules)</code>	Rules is the set of all step-equations.
<code>type_of(+Exp,?Type)</code>	Type is a reasonable guess for the type of Exp.
<code>theorem(?T,?G)</code>	The theorem named T has top-level goal G.
<code>wave_rule(+T,?N,?W,?Rn:Rule)</code>	Rn is the name of wave rule Rule with wave-term W, whose left hand side matches T in the Nth argument.
<code>wave_rules(?Rules)</code>	Rules is the set of all wave-equations.
<code>wave_occ(+T,?V,?P,?F,?S,?R)</code>	Variable V occurs in term T at position P, immediately surrounded by F, in a wave position of schema S, defined by rule R.

B The Tactics

The following is a list of the tactics that are available to CLAM together with a brief explanation of what they do.

<code>identity</code>	Finishes proof branches of the form $X=X$ in T .
<code>lemma(L)</code>	Finds a lemma L whose top-level goal is a universally quantified version of the current goal.
<code>backchain_lemma(L)</code>	Finds a lemma L that instantiates to $C \Rightarrow G$, with G the current goal and C a formula that occurs among the hypotheses.
<code>tautology(I)</code>	Find a series of inference rules I which prove that the current sequent is a propositional tautology.
<code>wave(Pos, Rule)</code>	Applies wave rule $Rule$ to position Pos in the current goal.
<code>base(Pos, Rule)</code>	Applies the base-equation $Rule$ to position Pos in the current goal.
<code>fertilize(Pos, Hyp)</code>	Finds an equality Hyp among the hypotheses with one side matching the subexpression at Pos in the current goal, and performs the corresponding substitutions.
<code>generalise(Exp)</code>	Replace a common subterm in both halves of an equality by a new universal variable.
<code>existential(Var:Type, Value)</code>	Replace an existentially quantified variable by a meta-variable.
<code>induction(Schema, Var:T)</code>	Apply induction according to $Schema$ on variable Var of type T .
<code>ind_strat(Schema, Var:T)</code>	Apply induction according to $Schema$ on variable Var , followed by a sequence of <code>base</code> 's in the base-case and a sequence of <code>wave</code> 's in the step-case, followed by a <code>fertilize</code> step.
<code>normalise(Normalisation)</code>	Apply a series of normalisation operations.
<code>sym_eval</code>	Iterate the following methods over the current sequent: <code>normalise</code> , <code>tautology</code> , <code>identity</code> , <code>step</code> and <code>base</code> .
<code>step(Pos, Rule)</code>	Applies the step-equation $Rule$ to position Pos in the current goal.

C The Definitions of the Object-Level Functions

The object-level functions used above were defined to have the following properties. Corresponding lemmas were used in the proofs, universally quantified. The two induction schemes shown, *primec* and *twos*, were proved as lemmas with the predicate variable ϕ universally quantified, and used to justify corresponding induction rules.

plus	$0 + y = y$ $s(x) + y = s(x + y)$
times	$0 \times y = 0$ $s(x) \times y = (x \times y) + x$
even	$even(0)$ $even(s(0)) \rightarrow void$ $even(s(s(x))) \leftrightarrow even(x)$
app	$app(nil, l) = l$ $app(h :: l1, l2) = h :: app(l1, l2)$
len	$len(nil) = 0$ $len(h :: l) = s(len(l))$
rev	$rev(nil) = nil$ $rev(h :: l) = app(rev(l), h :: nil)$
prod	$prod(nil) = s(0)$ $prod(h :: t) = h \times prod(t)$
primec	$\phi(0) \rightarrow \phi(s(0)) \rightarrow (\forall p. prime(p) \rightarrow \phi(p)) \rightarrow$ $(\forall x. \forall y. \phi(x) \rightarrow \phi(y) \rightarrow \phi(x \times y))$ $\rightarrow \forall z \phi(z)$
twos	$\phi(0) \rightarrow \phi(s(0)) \rightarrow (\forall x. \phi(x) \rightarrow \phi(s(s(x))))$ $\rightarrow \forall z \phi(z)$

References

- [Aubin 75] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes*. Institut de recherche d'informatique et d'automatique, 1975.
- [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 as DAI Research Paper No. 273.
- [Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
- [Bundy *et al* 88] 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. Also in the proceedings of IJCAI-89.
- [Burstall & Darlington 77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.
- [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 86] R.V. Desimone. Explanation-based learning of proof plans. In Y. Kodratoff, editor, *Proceedings of European Working Session on Learning, EWSL-86, Orsay, France*, February 1986. Longer version available from Edinburgh as Discussion Paper 6.
- [Gallaire & Lasserre 82] H. Gallaire and C. Lasserre. Metalevel control for logic programs. In K.L. Clark and S.-A. Tarnlund, editors, *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 Joerg Siekmann, editor, *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-Löf 79] Per Martin-Löf. 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.
- [Paulson 88] L. Paulson. Experience with Isabelle: A generic theorem prover. In *COLOG 88*. Institute of Cybernetics of the Estonian SSR, 1988.
- [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.