Andrew Ireland
November 9, 2001

A Guided Tour of the CLAM Proof Planner

1 Introduction

This note describes the CLAM proof planner. The aim is to provide a description that is sufficiently detailed so as to enable us to develop the architecture of a proof planning "wrapper" for the SPADE proof checker. While GUIs have been developed for CLAM [10, 7], my primary focus here is on the terminal based version. In particular I will draw upon the "critics version" of CLAM[1] because it is this version that provides the foundation for the CLAM-SPARK project.

2 The external perspective

CLAM is a Prolog based system with the user interacting at the level of the Prolog interpreter. Broadly speaking there are two modes of operation associated with CLAM, firstly the loading of theorem proving problems and secondly the actual planning of theorem proving problems. While both modes are essentially automatic, as will be described below, there are opportunities for user guidance.

2.1 Loading theorem proving problems

The basic philosophy behind the "proof planning" approach to theorem proving is that as information, *e.g.* definitions, theorems, etc, are presented to the system they are classified as to how they potentially might be used during the search for a proof. The majority of the code relating to loading theorem proving problems is located within `library.pl`. The main loading predicate is `lib_load` and the most general version takes the form `lib_load(A, B)`. Here `A` identifies a "logical object" and `B` identifies the directory were it is defined. The directory part is fairly straight forward, however, the logical object part needs further explanation. There are 16 kinds of logical object recognized by the library. I will consider 3, *i.e.* theorems, definitions and equations. Each logical object has a name (a Prolog identifier) and a wrapper, indicating the type of object it is. To illustrate, `thm(assm)` denotes a theorem (`thm`) called `assm` – associativity of multiplication for the natural numbers. Theorems in CLAM correspond to OYSTER theorems, which are essentially theorems expressed in the constructive logic of the NUPRL proof development environment [2]. The way in which `thm` objects are represented is covered in §3.4. Definitions within CLAM and OYSTER differ, this is reflected in the use of a `def` object to represent an OYSTER definition and `eqn` objects to represent CLAM level definitions. Basically an OYSTER definition is a syntactic abbreviations while a CLAM definition takes the form of a set of equations (typically recursive). In practical terms this means that loading a new function involves loading a `def` object together with a collection of `eqn` objects. This process is made easier by the fact that the library mechanism knows[2] which `eqn` objects to load when a `def` object is loaded. More generally, dependencies between `thm` objects and `def` objects can be recorded by what is called the "needs file" (`needs.pl`), *e.g.*

```
...
needs(def(times),          [def(plus)]).
...
needs(thm(assp),           [def(plus)]).
needs(thm(assm),           [def(times)]).
...
```

This records that the definition of the function `times` is defined in terms of `plus`, and that the theorems `assp` (associativity of plus) and `assm` (associativity of times) require the definitions of `plus` and `times` respectively. So the needs file provides a degree of automation when loading a theorem proving problem. However, one can always load objects individually as you please.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the CLAM-SPARK project (EPSRC GR/R24081).

[1]Internally known as version 3.2.

[2]The linkage is achieved by the use of a common naming convention, *e.g.* the definition of arithmetic addition involves a `def` object called `plus` and 2 `eqn` objects called `plus1` and `plus2`.

As well as supporting the loading of pre-classified objects, the library mechanism also supports a dynamic classification. The main example of this comes with *rippling* [1]. Rippling is implemented within CLAM by means of annotated rewrite rules called wave-rules. Wave-rules are derived automatically from a theorem by loading the theorem using a `wave` wrapper. However, every time a definition is loaded it is analyzed automatically to determine whether it provides wave-rules. This particular dynamic classification provides the meta-level context for a subsequent inductive proof planning attempt.

To illustrate the library mechanism at work the output generated by CLAM when loading the theorem `assm` follows:

```
| ?- lib_load(thm(assm)).
loading def(plus)...done
loading eqn(plus1)...done
loading eqn(plus2)...done
 adding wave-rules for plus1:..done
 adding wave-rules for plus2:.................done
 adding defeqn-record for plus1...done
 adding defeqn-record for plus2...done
loading def(times)...done
loading eqn(times1)...done
loading eqn(times2)...done
 adding wave-rules for times1:..done
 adding wave-rules for times2:......................done
 adding defeqn-record for times1...done
 adding defeqn-record for times2...done
loading thm(assm)...done
```

Note that as highlighted earlier, the needs file ensures that both the definitions of `plus` and `times` are loaded. Note also that after the `eqn` objects are loaded then the library attempts to automatically derive[3] wave-rules from the equations. The `defeqn-records` are not logical objects, rather they indicate the setting up of an efficient internal representation for equational definitions (more details in §3.4). Note that it is not until all the definitions are loaded that the actual theorem is loaded. The theorem as encoded within the OYSTER logic takes the form:

```
==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c)) = times(times(a,b),c) in pnat
```

where the type *pnat* denotes the natural numbers with Peano Arithmetic. For readers less familiar to constructive type theory the following presentation may be more accessible:

$$\forall a : pnat. \forall b : pnat. \forall c : pnat.\ times(a, times(b, c)) =_{pnat} times(times(a, b), c) \qquad (1)$$

Note that within constructive type theory equality is typed.

2.2 Planning a theorem proving problem

2.2.1 Plans as tactics

Plans generated by CLAM take the form of tactics [3], programs that control the application of proof rules within the context of a proof checker. As mentioned above, the CLAM proof checker is called OYSTER and its tactics correspond to the proof rules of the OYSTER logic[4]. In addition, OYSTER also provides the standard composition operators known as tacticals, *i.e.* `then`, `repeat`, `or`, `try` and `complete`. CLAM however, typically generates tactics that are constructed only using the `then` tactical. The general form of the `then` tactical is as follows:

```
Tac0 then [Tac1,...,TacN]
```

where `Tac0` is applied to the current goal generating `N` subgoals to which tactics `Tac1` through to `TacN` are applied respectively. Because CLAM deals with proof search while OYSTER deals with soundness, search oriented tacticals such as `repeat`, `or`, `try` and `complete` have no role to play within the tactics generated by CLAM.

In order to illustrate the form of a tactic produced by CLAM, I will consider again the `assm` example mentioned above. The tactic generated by CLAM for `assm` is as follows:

[3]The number of dots indicates the number of wave-rules that are derived from an equation.
[4]Each proof rule has a corresponding tactic, *i.e.* a rule inverse that supports backward style of proof.

```
induction([s(a)],[a:pnat]) then
   [eval_def([1,1],times1) then
      eval_def([1,2,1],times1) then
        eval_def([2,1],times1) then
          elementary(...),
    wave([1,2,1],[times2,equ(left)]) then
      wave([1,1],[times2,equ(left)]) then
        wave([2,1],[lemma1,equ(left)]) then
          fertilize(weak(v0)) then
            elementary(...)
   ]
```

This is a special purpose tactic, constructed to prove conjecture (1). Although special purpose it is constructed from a set of general purpose tactics, to be precise 5 general purpose tactics. These 5 tactics are the basic building blocks associated with CLAM's proof plan for mathematical induction. The first tactic, `induction`, applies a 1-step natural number induction on the variable `a`. This induction generates 2 subgoals, a base case and a step case. The base case part of the tactic involves simplification via the definition of `times`. This simplification corresponds to the sequence of 3 `eval_def` tactic applications. The base case is completed via the `elementary` tactic which is essentially a tautology checker augmented with basic properties of the underlying data types. In the step case part of the tactic a sequence of 3 `wave` tactic applications is specified followed by `fertilize` and `elementary`. Note that `wave` corresponds to rippling mentioned above while `fertilize` simply applies the induction hypothesis. Note also that all of the 5 general purpose tactics are parameterized, *e.g.* `wave([2,1],[lemma1,equ(left)])` specifies that the subterm at position `[2,1]` within the conclusion of the current goal is to be rewritten using the equational lemma called `lemma1` oriented left-to-right.

2.2.2 Plan construction

As mentioned above plan construction within CLAM corresponds to constructing a special purpose tactic for a given conjecture from a pre-defined set of general purpose tactics. Constructing such *customized tactics* involves search, *i.e.* searching for an appropriate composition of the basic tactics, which in turn involves searching for an appropriate instantiation for each basic tactic within the composition. Guiding this search is achieved through the use of tactic specifications called *methods*. For each of the basic tactics there exists a corresponding method. A method defines the context within which a particular tactic is applicable. Note that applicability encompasses both logical constraints as well as heuristic constraints. The process of plan construction can be thought of in terms of a "waterfall" where a goal is passed down an ordered sequence of methods. The sequence in the case of the induction proof plan is as follows:

```
elementary/1
eval_def/2
fertilize/1
wave/2
induction/2
```

Note that the weaker proof methods proceed the stronger proof methods. Once a tactic is judged to be applicable then the method is also used to compute the corresponding subgoals. In the case of (1) the strongest proof method, *i.e.* `induction`, is required. Feedback during the plan construction process takes the form of a proof script. To illustrate, the initial step in the `assm` proof planning script is given below:

```
assm([])
==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat
SELECTED METHOD at depth 0: induction([s(a)],[a:pnat])
|assm([2])
|a:pnat
|v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|==>b:pnat=>c:pnat=>times(``s({a})''<out>,times(\b/,\c/)) =
                         times(times(``s({a})''<out>,\b/),\c/)in pnat
```

The position of the goal within the overall goal tree is given by a list of numbers. In this case `[]`, which denotes the root node. Hypotheses and conclusion are separated as in NUPRL by the `==>` symbol. The method selected at this point in the planning process is `induction`. Note also that the induction hypothesis within the second subgoal is tagged, *i.e.* `ih(...)`. Goals are by default tackled using a depth first strategy. Iterative deepening, however, can

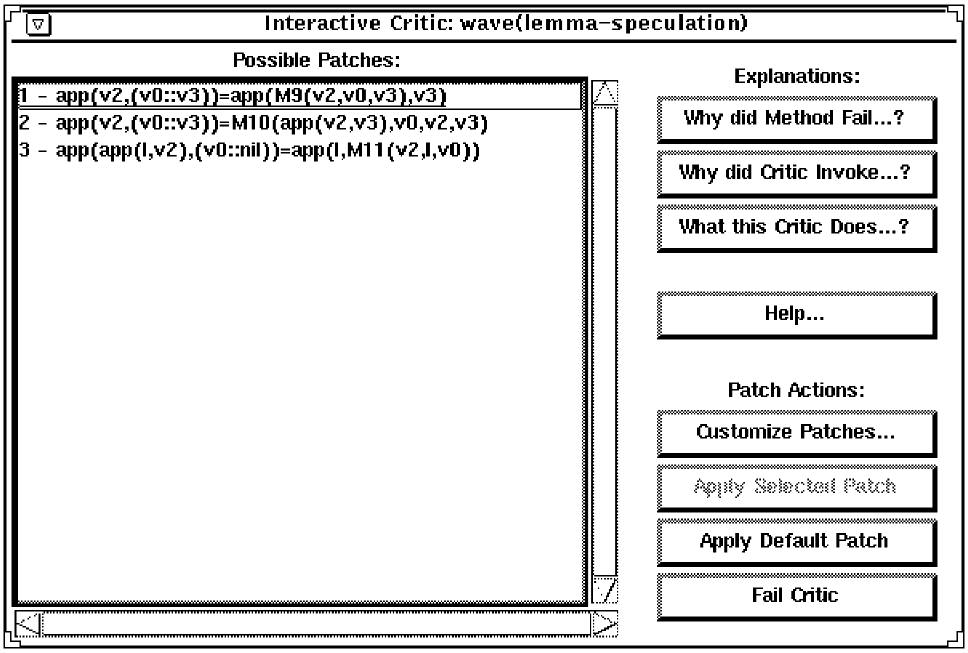

Figure 1: Example Lemma Speculation Patch Schemas

also be used (see §3.3). If N subgoals are generated at a particular point in the search then CLAM tackles subgoal N then subgoal $N-1$ down to subgoal 1. As illustrated above, this means that CLAM tackles step case subgoals before base case subgoals. The CLAM subgoaling process is fully automatic and generates a goal tree where the root of the tree corresponds to the initial conjecture and the leaves correspond to the application of what are referred to as terminating methods[5]. Once all the subgoals have been achieved then the customized tactic can be simply extracted from the goal tree. Note that the complete proof planning script for the `assm` example is given in the appendix.

2.2.3 Interactive proof critics

While proof planning was originally devised as an approach to automating theorem proving, investigations into interactive versions have been explored. From the perspective of adding a proof planning capability to SPADE, earlier work on *interactive proof critics* [7] may prove to be useful. Proof critics support proof patching via the analysis of proof planning failures. Discussion of how the mechanism works is delayed until §3.3. However, the technique has been used successfully in generating missing lemmas, generalizing inductive conjectures, finding case splits, generating loop invariants, revising the selection of induction rules [4, 5, 6, 8, 9, 11, 12]. In general, for a given proof failure, a number of proof critics may be applicable. Moreover, each proof critic may give rise to a number of distinct patch schemas. In such situations a user may usefully prioritize the "more promising" candidates and thus reduce the search. In addition, a user may be able to short-circuit the proof planner's incremental instantiation of a patch schema, either partially or fully. This was the motivation for exploring interactive proof critics. The resulting interface is illustrated in figure 1 where multiple patch schemas, in this case schematic lemmata, are presented to the user. Here if the user provides the appropriate instantiation for the default patch schema then the subsequent proof planning automatically discovers the required lemmata.

2.2.4 Plan execution

Plan execution corresponds to tactic execution within OYSTER. Because our target proof checker is SPADE and not OYSTER detail of the tactic evaluation mechanism is not relevant here.

[5]Methods that generate no subgoals.

3 The internal perspective

3.1 Plan construction

As mentioned above, the key building blocks within the proof planning process are proof methods. The general form of a method is as follows:

```
method(Name(Args),
       Goal,
       Preconditions,
       Effects,
       Subgoals,
       Substitutions
      ).
```

A method has 6 slots, the first slot identifies the method, *i.e.* `Name` and specifies a set of arguments `Args`. The role of the arguments is explained below. The second slot denotes the current goal while the third slot provides preconditions that define the applicability of the associated tactic. The preconditions are defined in terms of a set of meta-logical predicates, *i.e.* predicates that describe properties of the underlying logic. The preconditions define what must be true within the current proof context[6] for the associated tactic to be applicable. Note that the arguments (slot 1) associated with a method will be referenced within the preconditions, the instantiation of the arguments occurs as a side-effect of precondition evaluation. If all the preconditions succeed then the `Effects` are used to compute the form of the `Subgoals`. While method preconditions are declarative, effects are more procedural. The final slot is used to store substitutions for higher-order meta variables, these are used during *proof patching* which is detailed in section §3.3.

A concrete example of a method, *i.e.* `wave`, is presented below.

```
method(wave(P,[Rule,Dir]),
       H==>G,
       [wave_occ_at(G,P,L),
        wave_rule_match(Rule,long(Dir),C=>L:=>R,SubstL),
        trivial(H==>C)
       ],
       [replace_in_matrix(P,R,G,NG),
        sink_expansions(NG,NewG)
       ],
       [H==>NewG],
       SubstL
      ).
```

The `wave` method implements rippling, a restricted form of rewriting in which a heuristic measure is used to reduce syntactic differences between formulae. This is achieved by adding "meta-data" to a goal that highlights which parts of the induction conclusion differ syntactically from the induction hypothesis. Restricting the rewriting of the induction conclusion so as to eliminate the differences is achieved by the application of wave-rules and it is the `wave` method that controls the application of wave-rules. Note that the method arguments, *i.e.* `P`, `Rule` and `Dir`, are all referenced within the preconditions to the method. As a consequence, if all the preconditions succeed within a particular context then as a side-effect the arguments are instantiated.

In order to show a concrete application of the `wave` method I will consider again the proof of the `assm` conjecture. In particular, consider the middle of the step case proof where the induction hypothesis takes the form:

```
v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c) in pnat)
```

and the induction conclusion takes the form:

```
b:pnat=>c:pnat=>
      ``plus({times(a,times(\b/,\c/))},times(\b/,\c/))''<out> =
                    times(``plus({times(a,\b/)},\b/)''<out>,\c/) in pnat
```

Making progress requires eliminating the occurrence of `plus(...,b)` from the right-hand side of the equation since it is preventing a match with the right-hand side of the induction hypothesis. The annotations, or meta-data, embedded within the formula highlight which parts of the formula need to be eliminated. Evaluating the preconditions of the `wave` method within the context of this goal gives rise to the following instantiation:

[6] Current goal together with its associated theory, *i.e.* definitions, properties *etc.*

1. wave_occ_at(G,P,L): there exists a wave occurrence L at position P within the goal G.

   G: ‘‘plus({times(a,times(\b/,\c/))},times(\b/,\c/))’’<out> =

   times(‘‘plus({times(a,\b/)},\b/)’’<out>,\c/) in pnat

   P: [2,1]

   L: times(‘‘plus({times(a,\b/)},\b/)’’<out>,\c/))

2. wave_rule_match(Rule,long(Dir),C=>L:=>R,SubstL): there exists a wave-rule that matches L.

   Rule: lemma1

   Dir: equ(left)

   C: []

   L: times(‘‘plus({times(a,\b/)},\b/)’’<out>,\c/))

   R: ‘‘plus({times(times(a,\b/),\c/)},times(\b/,\c/))’’<out>

   SubstL: []

3. trivial(H==>C): any condition attached to the wave-rule follows from the hypothesis list H.

   H: Hypothesis list

   C: []

The effect of the method is to rewrite the subterm:

times(plus(times(a,b),b),c)

using lemma1 (oriented left-to-right):

times(plus(X,Y),Z) :=> plus(times(X,Z),times(Y,Z))

giving rise to a new subterm of the form:

plus(times(times(a,b),c),times(b,c))

Note that all the information required to perform this rewrite step, *i.e.* the position of the subterm within the formula, the lemma and orientation, is encoded within the arguments of the method:

wave([2,1],[lemma1,equ(left)])

These arguments are shared by the corresponding wave tactic, as a consequence tactic application requires no search.

3.2 Plan execution

Although the OYSTER tactic application mechanism is not relevant to the theme of this note, it is worth noting that the “basic tactics” corresponding to CLAM methods are not basic OYSTER tactics. To illustrate, below is shown some of the Prolog code for the wave tactic:

```
wave(Pos,[Rule,Dir]) :-
    dequantify_all(Vars,Frees)
       then rewrite_at_pos(Pos,Rule,Dir)
          then requantify(Vars,Frees).

dequantify_all(Vars,Frees) :-
    goal(G), matrix(Vars,_,G), dequantify(Vars,Frees).

rewrite_at_pos(Pos,L=R in T,equ(left)) :-
    goal(G),hyp_list(Hl), (hypothesis(Eq:L=R in T) ; hypothesis(Eq:R=L in T)),
    genvar(V), free([V],Hl), \+ appears(V,G), \+ appears(V,R),
    replace(Pos,V,G,Gv),
    top_univ_level(G,I),II is I+1,
    subst(at(II),over(V,Gv),L=R in T)
       then [univ_elim(Eq), idtac,wfftacs].
rewrite_at_pos(Pos,H,Dir) :-
    hypothesis(H:X=Y in T),!,
    rewrite_at_pos(Pos,X=Y in T,Dir).
rewrite_at_pos(Pos,H,Dir) :-
    hypothesis(H:_=>X=Y in T), !,
```

```
    hyp_list(HL), free([EqH],HL),
    elim(H,new[EqH])
     then [elementary,
           rewrite_at_pos(Pos,X=Y in T,Dir) then thin([EqH])].

requantify([],[]) :- !.
requantify(Vars,Frees) :-
    goal(G), untype(Vars,Vs), s(G,Vs,Frees,NewG), quantify(Vars,NewG,QNewG),
    hyp_list(H), free([Seq],H),
    seq(QNewG,new[Seq])
        then [thin(Frees),
              elim_on(Seq,Frees,Last)
                  then hyp(Last)].
```

The aim of showing such detail is to highlight to the reader the leverage that proof planning can bring, *i.e.* 1 planning step will typically unpack into between 10 and 20 proof checking steps.

3.3 Proof patching

Methods may have associated proof critics. When a method fails to apply then each of the associated proof critics is tried in turn. When the preconditions of a critic succeed then the associated patch is applied. If none of the critics are applicable then the next method in the waterfall of methods is evaluated. Backtracking over both method and critic applications is possible. When in proof patching mode an iterative deepening search strategy is typically used. The general form of a proof critic is as follows:

```
critic(Name(Args),
       Plan,
       Prep,
       Preconditions,
       Patch
      ).
```

The first slot identifies the critic and its arguments. The second slot identifies the current proof planning effort while the third slot provides the ability to perform calculations in preparation for the precondition evaluation (slot 4). Note that precondition evaluation here typically involves analyzing partial success at the level of method precondition evaluation. The final slot is used to generate the proof patch, or at least setup a schematic proof attempt that will lead to a complete patch. In the case of the `assm` example the proof patching capabilities of CLAM result in the automatic generation of 3 lemmas during the search for the proof of the main conjecture:

```
lemma1: [b:pnat,c:pnat,v0:pnat] |- times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat

lemma2: [c:pnat,v1:pnat,v2:pnat] |- plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat

lemma3: [c:pnat,v1:pnat] |- plus(v1,s(c))=s(plus(v1,c))in pnat
```

For more details see the appendix. The `lemma-calculation` critic used in generating the missing lemmas is given below:

```
critic(wave(lemma-calculation),
       Plan,
       [current_node(Plan, Addr),
        node(Plan, Addr, H==>G, _, _, _, Preconds, _),
        construct_context(H, G, Context)],
       [not contains_meta_variables(G),
        member([wave_occ_at(_, _, _),
                false,
                false|
                _], Preconds),
        partially_blocked(G)],
       [construct_lemma_lhs(H==>G, LHS, Typ),
        construct_lemma_rhs(H==>G, RHS, Typ),
        construct_generalized_lemma(LHS, RHS, Typ, Context, Name:Lemma),
        validate_calculation(Plan, Name:Lemma)]
      ).
```

The key intuition behind this proof patch is the fact that there is a missing wave-rule and one side of the equational conclusion can be matched against the induction hypothesis (`partially_blocked(G)`). This means that the induction hypothesis can be used in conjunction with the induction conclusion to suggest the form of the missing lemma. Note that in the case of `lemma-calculation` only first-order meta-variables are required, in general however proof patching requires the use of higher-order meta-variables in providing place-holders for missing information. This is the reason for the `SubstL` slot within the method template.

3.4 Representational issues

The internal representation of the various logical objects supported by CLAM is achieved through Prolog's record database. For instance, a `theorem` is recorded in the following form:

```
recorded(theorem, theorem(Name, Type, Goal, Thm), Ref)
```

where `Name` is the name of the logical object, `Type` the type of logical object (`Type(Name)`) and `Goal` the top-level goal of the logical object. Typically, `Thm` will be the same as `Name`, except for recursive equations where `Name` will also include a number indicating which member of the recursive set it is. Similar structures are used for storing meta-level objects such as wave-rules, *i.e.*

```
recorded(wave, wave(Name, Rule, Type), Ref)
```

Note that additional access predicates are built upon these low-level representations.

In terms of the actual proof planning process, CLAM's subgoaling mechanism plays a central role. As mentioned above, CLAM mmaintains a goal tree during the search for a successful proof plan instantiation. The goal tree is represented as a set of nodes. To illustrate, consider again the snap-shot taken from the `assm` proof that was presented in §2.2.2:

```
assm([])
==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat
SELECTED METHOD at depth 0: induction([s(a)],[a:pnat])
|assm([2])
|a:pnat
|v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|==>b:pnat=>c:pnat=>times(``s({a})''<out>,times(\b/,\c/)) =
                            times(times(``s({a})''<out>,\b/),\c/)in pnat
```

Each node is represented as an asserted fact constructed by the predicate `node/8`, *i.e.*

```
node(PlanName, Address, Sequent, Status, Method, Substs, Preconds, Count)
```

The relationship between this particular `assm` node and this general node template is outlined below:

1. `PlanName`: the name of the conjecture being planned.

   `assm`

2. `Address`: the address of the node within the node tree.

   `[]`

3. `Sequent`: the conjecture.

   `[]==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat`

4. `Status`: the status of the conjecture, *i.e.* `open` (leaf node) or `closed` (internal node).

   `closed`

5. `Method`: the chosen method.

   `induction([s(a)],[a:pnat])`

6. `Substs`: list of substitutions generated for higher-order meta variables which appear within the conjecture.

   `[]`

7. `Preconds`: the set of instantiated preconditions, including partial successes.

   `[[wave_occ([],a:pnat=>b:pnat=>c:pnat=> ...in...,[b:pnat],[s(_A)])]]`

8. Count: the number of subgoals resulting from the method application.

   2

Note that the Preconds slot is crucial for proof patching since a record of the partial success of a method's preconditions underpins the proof critics failure analysis mechanism.

4 Adapting proof planning for new applications

Below I present a first attempt at a "shopping list" of the aspects of CLAM that are OYSTER dependent and will need to be changed for SPADE. Note that I focus on what potentially needs to change rather than on how it should be changed:

- The structure of definitions, conjectures, theorems, rewrite rules will need to be re-aligned with the SPADE logic.
- The need for the proof planning wrapper to have a separate library mechanism will need to be considered.
- The exact nature of the meta-level classification process that underlies the basic proof planning "philosophy" will need to be considered.
- The meta-logical predicates used within method and critic preconditions will need to be re-aligned with the SPADE logic.
- A replacement for OYSTER level tactics will be required.
- The level at which interaction between the proof planning wrapper and SPADE will need to be made precise.
- The level of interaction between the target user and the proof planning wrapper will need to be made precise.

5 Conclusion

This note presents a rough guide to the CLAM proof planner and complements CSnote 4 where the SPADE perspective is detailed. The aim of these notes is to facilitate a discussion as to how best a proof planning capability can be added to the SPADE proof checker. A subsequent note will report on this discussion and provide our blueprint of a proof planning wrapper for SPADE.

Appendix: Poof planning script for assm problem

```
assm([])
==>a:pnat=>b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat
SELECTED METHOD at depth 0: induction([s(a)],[a:pnat])
|assm([2])
|a:pnat
|v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|==>b:pnat=>c:pnat=>times(``s({a})''<out>,times(\b/,\c/))=times(times(``s({a})''<out>,\b/),\c/)in pnat
|SELECTED METHOD at depth 1: wave([1,2,1],[times2,equ(left)])
||assm([1,2])
||a:pnat
||v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
||==>b:pnat=>c:pnat=>times(``s({a})''<out>,times(\b/,\c/))=times(``plus({times(a,\b/)},\b/)''<out>,\c/)in pnat
||SELECTED METHOD at depth 2: wave([1,1],[times2,equ(left)])
|||assm([1,1,2])
|||a:pnat
|||v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|||==>b:pnat=>c:pnat=>``plus({times(a,times(\b/,\c/))},times(\b/,\c/))''<out>=
                                      times(``plus({times(a,\b/)},\b/)''<out>,\c/)in pnat

>>>>> INVOKING lemma-calculation CRITIC <<<<<

------------------------------------------------------------
lemma1:

[b:pnat,c:pnat,v0:pnat]

|- times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat
```

```
saving thm(lemma1)...done
lemma1([])
==>b:pnat=>c:pnat=>v0:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat
SELECTED METHOD at depth 0: induction([s(v0)],[v0:pnat])
|lemma1([2])
|v0:pnat
|v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
|==>b:pnat=>c:pnat=>times(plus(``s({v0})''<out>,\b/),\c/)=plus(times(``s({v0})''<out>,\c/),times(\b/,\c/))in pnat
|SELECTED METHOD at depth 1: wave([1,2,1],[times2,equ(left)])
||lemma1([1,2])
||v0:pnat
||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
||==>b:pnat=>c:pnat=>times(plus(``s({v0})''<out>,\b/),\c/)=
                           plus(``plus({times(v0,\c/)},\c/)''<out>,times(\b/,\c/))in pnat
||SELECTED METHOD at depth 2: wave([1,1,1],[plus2,equ(left)])
|||lemma1([1,1,2])
|||v0:pnat
|||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
|||==>b:pnat=>c:pnat=>times(``s({plus(v0,\b/)})''<out>,\c/)=
                                 plus(``plus({times(v0,\c/)},\c/)''<out>,times(\b/,\c/))in pnat
|||SELECTED METHOD at depth 3: wave([1,1],[times2,equ(left)])
||||lemma1([1,1,1,2])
||||v0:pnat
||||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
||||==>b:pnat=>c:pnat=>``plus({times(plus(v0,\b/),\c/)},\c/)''<out>=
                                     plus(``plus({times(v0,\c/)},\c/)''<out>,times(\b/,\c/))in pnat

>>>>> INVOKING lemma-calculation CRITIC <<<<<

------------------------------------------------------------
lemma2:

[c:pnat,v1:pnat,v2:pnat]

|- plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat

saving thm(lemma2)...done
lemma2([])
==>c:pnat=>v1:pnat=>v2:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat
SELECTED METHOD at depth 0: induction([s(v2)],[v2:pnat])
|lemma2([2])
|v2:pnat
|v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
|==>c:pnat=>v1:pnat=>plus(plus(``s({v2})''<out>,\c/),\v1/)=plus(plus(``s({v2})''<out>,\v1/),\c/)in pnat
|SELECTED METHOD at depth 1: wave([1,2,1],[plus2,equ(left)])
||lemma2([1,2])
||v2:pnat
||v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
||==>c:pnat=>v1:pnat=>plus(plus(``s({v2})''<out>,\c/),\v1/)=plus(``s({plus(v2,\v1/)})''<out>,\c/)in pnat
||SELECTED METHOD at depth 2: wave([1,1,1],[plus2,equ(left)])
|||lemma2([1,1,2])
|||v2:pnat
|||v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
|||==>c:pnat=>v1:pnat=>plus(``s({plus(v2,\c/)})''<out>,\v1/)=plus(``s({plus(v2,\v1/)})''<out>,\c/)in pnat
|||SELECTED METHOD at depth 3: wave([2,1],[plus2,equ(left)])
||||lemma2([1,1,1,2])
||||v2:pnat
||||v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
||||==>c:pnat=>v1:pnat=>plus(``s({plus(v2,\c/)})''<out>,\v1/)=``s({plus(plus(v2,\v1/),\c/)})''<out> in pnat
||||SELECTED METHOD at depth 4: wave([1,1],[plus2,equ(left)])
|||||lemma2([1,1,1,1,2])
|||||v2:pnat
|||||v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
|||||==>c:pnat=>v1:pnat=>``s({plus(plus(v2,\c/),\v1/)})''<out>=``s({plus(plus(v2,\v1/),\c/)})''<out> in pnat
|||||SELECTED METHOD at depth 5: fertilize(weak(v0))
||||||lemma2([1,1,1,1,1,2])
||||||v2:pnat
||||||v0:ih(c:pnat=>v1:pnat=>plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat)
||||||==>c:pnat=>v1:pnat=>s(plus(plus(v2,c),v1))=s(plus(plus(v2,c),v1))in pnat
||||||TERMINATING METHOD at depth 6: elementary(...)
|lemma2([1])
|v2:pnat
|==>c:pnat=>v1:pnat=>plus(plus(0,c),v1)=plus(plus(0,v1),c)in pnat
|SELECTED METHOD at depth 1: eval_def([1,2,1],plus1)
```

```
||lemma2([1,1])
||v2:pnat
||==>c:pnat=>v1:pnat=>plus(plus(0,c),v1)=plus(v1,c)in pnat
||SELECTED METHOD at depth 2: eval_def([1,1,1],plus1)
|||lemma2([1,1,1])
|||v2:pnat
|||==>c:pnat=>v1:pnat=>plus(c,v1)=plus(v1,c)in pnat
|||SELECTED METHOD at depth 3: induction([s(c)],[c:pnat])
||||lemma2([2,1,1,1])
||||c:pnat
||||v0:ih(v1:pnat=>plus(c,v1)=plus(v1,c)in pnat)
||||v2:pnat
||||==>v1:pnat=>plus(``s({c})''<out>,\v1/)=plus(\v1/,``s({c})''<out>)in pnat
||||SELECTED METHOD at depth 4: wave([1,1],[plus2,equ(left)])
|||||lemma2([1,2,1,1,1])
|||||c:pnat
|||||v0:ih(v1:pnat=>plus(c,v1)=plus(v1,c)in pnat)
|||||v2:pnat
|||||==>v1:pnat=>``s({plus(c,\v1/)})''<out>=plus(\v1/,``s({c})''<out>)in pnat

>>>>> INVOKING lemma-calculation CRITIC <<<<<

------------------------------------------------------------
lemma3:

[c:pnat,v1:pnat]

|- plus(v1,s(c))=s(plus(v1,c))in pnat

saving thm(lemma3)...done
lemma3([])
==>c:pnat=>v1:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat
SELECTED METHOD at depth 0: induction([s(v1)],[v1:pnat])
|lemma3([2])
|v1:pnat
|v0:ih(c:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat)
|==>c:pnat=>plus(``s({v1})''<out>,s(\c/))=s(plus(``s({v1})''<out>,\c/))in pnat
|SELECTED METHOD at depth 1: wave([1,2,1],[plus2,equ(left)])
||lemma3([1,2])
||v1:pnat
||v0:ih(c:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat)
||==>c:pnat=>plus(``s({v1})''<out>,s(\c/))=s(``s({plus(v1,\c/)})''<out>)in pnat
||SELECTED METHOD at depth 2: wave([1,1],[plus2,equ(left)])
|||lemma3([1,1,2])
|||v1:pnat
|||v0:ih(c:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat)
|||==>c:pnat=>``s({plus(v1,s(\c/))})''<out>=s(``s({plus(v1,\c/)})''<out>)in pnat

>>>>> INVOKING meta-ripple-position CRITIC <<<<<

|||lemma3([1,1,2])
|||v1:pnat
|||v0:ih(c:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat)
|||==>c:pnat=>``s({plus(v1,s(\c/))})''<out>=``s({s(plus(v1,\c/))})''<out> in pnat
|||SELECTED METHOD at depth 3: fertilize(weak(v0))
||||lemma3([1,1,1,2])
||||v1:pnat
||||v0:ih(c:pnat=>plus(v1,s(c))=s(plus(v1,c))in pnat)
||||==>c:pnat=>s(plus(v1,s(c)))=s(plus(v1,s(c)))in pnat
||||TERMINATING METHOD at depth 4: elementary(...)
|lemma3([1])
|v1:pnat
|==>c:pnat=>plus(0,s(c))=s(plus(0,c))in pnat
|SELECTED METHOD at depth 1: eval_def([1,1],plus1)
||lemma3([1,1])
||v1:pnat
||==>c:pnat=>s(c)=s(plus(0,c))in pnat
||SELECTED METHOD at depth 2: eval_def([1,2,1],plus1)
|||lemma3([1,1,1])
|||v1:pnat
|||==>c:pnat=>s(c)=s(c)in pnat
|||TERMINATING METHOD at depth 3: elementary(...)
 adding wave-rules for lemma3:................done
|||||lemma2([1,2,1,1,1])
```

```
|||||c:pnat
|||||v0:ih(v1:pnat=>plus(c,v1)=plus(v1,c)in pnat)
|||||v2:pnat
|||||==>v1:pnat=>``s({plus(c,\v1/)})''<out>=plus(\v1/,``s({c})''<out>)in pnat
|||||SELECTED METHOD at depth 5: wave([2,1],[lemma3,equ(left)])
||||||lemma2([1,1,2,1,1,1])
||||||c:pnat
||||||v0:ih(v1:pnat=>plus(c,v1)=plus(v1,c)in pnat)
||||||v2:pnat
||||||==>v1:pnat=>``s({plus(c,\v1/)})''<out>=``s({plus(\v1/,c)})''<out> in pnat
||||||SELECTED METHOD at depth 6: fertilize(weak(v0))
|||||||lemma2([1,1,1,2,1,1,1])
|||||||c:pnat
|||||||v0:ih(v1:pnat=>plus(c,v1)=plus(v1,c)in pnat)
|||||||v2:pnat
|||||||==>v1:pnat=>s(plus(c,v1))=s(plus(c,v1))in pnat
|||||||TERMINATING METHOD at depth 7: elementary(...)
||||lemma2([1,1,1,1])
||||c:pnat
||||v2:pnat
||||==>v1:pnat=>plus(0,v1)=plus(v1,0)in pnat
||||SELECTED METHOD at depth 4: eval_def([1,1],plus1)
|||||lemma2([1,1,1,1,1])
|||||c:pnat
|||||v2:pnat
|||||==>v1:pnat=>v1=plus(v1,0)in pnat
|||||SELECTED METHOD at depth 5: induction([s(v1)],[v1:pnat])
||||||lemma2([2,1,1,1,1,1])
||||||v1:pnat
||||||v0:ih(v1=plus(v1,0)in pnat)
||||||c:pnat
||||||v2:pnat
||||||==>``s({v1})''<out>=plus(``s({v1})''<out>,0)in pnat
||||||SELECTED METHOD at depth 6: wave([2,1],[plus2,equ(left)])
|||||||lemma2([1,2,1,1,1,1,1])
|||||||v1:pnat
|||||||v0:ih(v1=plus(v1,0)in pnat)
|||||||c:pnat
|||||||v2:pnat
|||||||==>``s({v1})''<out>=``s({plus(v1,0)})''<out> in pnat
|||||||SELECTED METHOD at depth 7: fertilize(weak(v0))
||||||||lemma2([1,1,2,1,1,1,1,1])
||||||||v1:pnat
||||||||v0:ih(v1=plus(v1,0)in pnat)
||||||||c:pnat
||||||||v2:pnat
||||||||==>s(v1)=s(v1)in pnat
||||||||TERMINATING METHOD at depth 8: elementary(...)
||||||lemma2([1,1,1,1,1,1])
||||||v1:pnat
||||||c:pnat
||||||v2:pnat
||||||==>0=plus(0,0)in pnat
||||||SELECTED METHOD at depth 6: eval_def([2,1],plus1)
|||||||lemma2([1,1,1,1,1,1,1])
|||||||v1:pnat
|||||||c:pnat
|||||||v2:pnat
|||||||==>0=0 in pnat
|||||||TERMINATING METHOD at depth 7: elementary(...)
 adding wave-rules for lemma2:......done
||||lemma1([1,1,1,2])
||||v0:pnat
||||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
||||==>b:pnat=>c:pnat=>``plus({times(plus(v0,\b/),\c/)},\c/)''<out>=
                              plus(``plus({times(v0,\c/)},\c/)''<out>,times(\b/,\c/))in pnat
||||SELECTED METHOD at depth 4: wave([2,1],[lemma2,equ(left)])
|||||lemma1([1,1,1,1,2])
|||||v0:pnat
|||||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
|||||==>b:pnat=>c:pnat=>``plus({times(plus(v0,\b/),\c/)},\c/)''<out>=
                              ``plus({plus(times(v0,\c/),times(\b/,\c/))},\c/)''<out> in pnat
|||||SELECTED METHOD at depth 5: fertilize(weak(v1))
||||||lemma1([1,1,1,1,1,2])
```

```
||||||v0:pnat
||||||v1:ih(b:pnat=>c:pnat=>times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat)
||||||==>b:pnat=>c:pnat=>plus(times(plus(v0,b),c),c)=plus(times(plus(v0,b),c),c)in pnat
||||||TERMINATING METHOD at depth 6: elementary(...)
|lemma1([1])
|v0:pnat
|==>b:pnat=>c:pnat=>times(plus(0,b),c)=plus(times(0,c),times(b,c))in pnat
|SELECTED METHOD at depth 1: eval_def([1,2,1],times1)
||lemma1([1,1])
||v0:pnat
||==>b:pnat=>c:pnat=>times(plus(0,b),c)=plus(0,times(b,c))in pnat
||SELECTED METHOD at depth 2: eval_def([2,1],plus1)
|||lemma1([1,1,1])
|||v0:pnat
|||==>b:pnat=>c:pnat=>times(plus(0,b),c)=times(b,c)in pnat
|||SELECTED METHOD at depth 3: eval_def([1,1,1],plus1)
||||lemma1([1,1,1,1])
||||v0:pnat
||||==>b:pnat=>c:pnat=>times(b,c)=times(b,c)in pnat
||||TERMINATING METHOD at depth 4: elementary(...)
 adding wave-rules for lemma1:............................................................done
|||assm([1,1,2])
|||a:pnat
|||v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|||==>b:pnat=>c:pnat=>``plus({times(a,times(\b/,\c/))},times(\b/,\c/))''<out>=
                              times(``plus({times(a,\b/)},\b/)''<out>,\c/)in pnat
|||SELECTED METHOD at depth 3: wave([2,1],[lemma1,equ(left)])
||||assm([1,1,1,2])
||||a:pnat
||||v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
||||==>b:pnat=>c:pnat=>``plus({times(a,times(\b/,\c/))},times(\b/,\c/))''<out>=
                               ``plus({times(times(a,\b/),\c/)},times(\b/,\c/))''<out> in pnat
||||SELECTED METHOD at depth 4: fertilize(weak(v0))
|||||assm([1,1,1,1,2])
|||||a:pnat
|||||v0:ih(b:pnat=>c:pnat=>times(a,times(b,c))=times(times(a,b),c)in pnat)
|||||==>b:pnat=>c:pnat=>plus(times(a,times(b,c)),times(b,c))=plus(times(a,times(b,c)),times(b,c))in pnat
|||||TERMINATING METHOD at depth 5: elementary(...)
|assm([1])
|a:pnat
|==>b:pnat=>c:pnat=>times(0,times(b,c))=times(times(0,b),c)in pnat
|SELECTED METHOD at depth 1: eval_def([1,1],times1)
||assm([1,1])
||a:pnat
||==>b:pnat=>c:pnat=>0=times(times(0,b),c)in pnat
||SELECTED METHOD at depth 2: eval_def([1,2,1],times1)
|||assm([1,1,1])
|||a:pnat
|||==>b:pnat=>c:pnat=>0=times(0,c)in pnat
|||SELECTED METHOD at depth 3: eval_def([2,1],times1)
||||assm([1,1,1,1])
||||a:pnat
||||==>b:pnat=>c:pnat=>0=0 in pnat
||||TERMINATING METHOD at depth 4: elementary(...)
Planning complete for assm

------------------------------------------------------------
assm:

[a:pnat,b:pnat,c:pnat]

|- times(a,times(b,c))=times(times(a,b),c)in pnat

induction([s(a)],[a:pnat]) then
  [eval_def([1,1],times1) then
     eval_def([1,2,1],times1) then
       eval_def([2,1],times1) then
         elementary(...),
   wave([1,2,1],[times2,equ(left)]) then
     wave([1,1],[times2,equ(left)]) then
       wave([2,1],[lemma1,equ(left)]) then
         fertilize(weak(v0)) then
           elementary(...)
  ]
```

```
------------------------------------------------------------
lemma1:

[b:pnat,c:pnat,v0:pnat]

|- times(plus(v0,b),c)=plus(times(v0,c),times(b,c))in pnat

induction([s(v0)],[v0:pnat]) then
  [eval_def([1,2,1],times1) then
     eval_def([2,1],plus1) then
       eval_def([1,1,1],plus1) then
         elementary(...),
   wave([1,2,1],[times2,equ(left)]) then
     wave([1,1,1],[plus2,equ(left)]) then
       wave([1,1],[times2,equ(left)]) then
         wave([2,1],[lemma2,equ(left)]) then
           fertilize(weak(v1)) then
             elementary(...)
  ]

------------------------------------------------------------
lemma2:

[c:pnat,v1:pnat,v2:pnat]

|- plus(plus(v2,c),v1)=plus(plus(v2,v1),c)in pnat

induction([s(v2)],[v2:pnat]) then
  [eval_def([1,2,1],plus1) then
     eval_def([1,1,1],plus1) then
       induction([s(c)],[c:pnat]) then
         [eval_def([1,1],plus1) then
            induction([s(v1)],[v1:pnat]) then
              [eval_def([2,1],plus1) then
                 elementary(...),
               wave([2,1],[plus2,equ(left)]) then
                 fertilize(weak(v0)) then
                   elementary(...)
              ],
          wave([1,1],[plus2,equ(left)]) then
            wave([2,1],[lemma3,equ(left)]) then
              fertilize(weak(v0)) then
                elementary(...)
         ],
   wave([1,2,1],[plus2,equ(left)]) then
     wave([1,1,1],[plus2,equ(left)]) then
       wave([2,1],[plus2,equ(left)]) then
         wave([1,1],[plus2,equ(left)]) then
           fertilize(weak(v0)) then
             elementary(...)
  ]

------------------------------------------------------------
lemma3:

[c:pnat,v1:pnat]

|- plus(v1,s(c))=s(plus(v1,c))in pnat

induction([s(v1)],[v1:pnat]) then
  [eval_def([1,1],plus1) then
     eval_def([1,2,1],plus1) then
       elementary(...),
   wave([1,2,1],[plus2,equ(left)]) then
     wave([1,1],[plus2,equ(left)]) then
       fertilize(weak(v0)) then
         elementary(...)
  ]
yes
| ?-
```

References

[1] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

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

[3] 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.

[4] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[5] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[6] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[7] A. Ireland, M. Jackson, and G. Reid. Interactive Proof Critics. *Formal Aspects of Computing: The International Journal of Formal Methods*, 11(3):302–325, 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/15.

[8] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[9] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[10] H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag, 1997.

[11] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998. An earlier version is available from the Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/2.

[12] J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In *The* 14^{th} *IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE Computer Society, 1999.