CS-Note[*] 21

Bill J. Ellis
September 12, 2002

Raising NuSPADE Issues (Volume 3)

Contents

[*]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 NuSPADE project (EPSRC GR/R24081).

1 Introduction

See CS-note 19 for introduction.

2 Managing Hypothesis

The PLANNER works on a goal containing a number of hypothesis and a single conclusion. This fits in with the organisation of planning outlined in CS-note 20 section 3. In the PLANNER it is considered necessary to manage hypothesis in a stronger manner than that undertaken by CLAM.

2.1 CLAM Hypothesis

A number of important features about the treatment of hypothesis in CLAM are identified:

- **Keep hypothesis local -** During the construction of a proof plan a tree is created, with the initial problem at the root. Each node of this tree contains the current problem as `[Hypothesis]==>Conclusion`. Thus the hypothesis of CLAM are always local to each node in the proof tree.

  It is necessary for some hypothesis to be local. For example, to hold different cases after a case split. However, some hypothesis could logically be global to the entire plan. These are the initial hypothesis available at the root node, plus any further hypothesis that can be derived from these. In particular, any conjunct proved of the initial conclusion.

- **Names as OYSTER with extra annotations -** The hypothesis at each node are assigned a name that exactly corresponds with the equivalent OYSTER name for this point in the proof. Further, significant hypothesis are annotated so they can be referred to from within a proof. For example an induction hypothesis is marked in amongst the hypothesis using:

  ```
  [..., (Oyster-variable:ih(Induction-hypothesis)), ... ]
  ```

- **Cary around syntax and names -** The local hypothesis held at each node have their name and structure copied entirety. Having the full structure of the hypothesis available is beneficial in exploiting matching in the underlying Prolog implementation. The name is required during the application of methods to record which hypothesis were used.

2.2 Proposal: Names as Pointer To Global Store

There are a number of problems and restrictions presented by the current CLAM method for storing hypothesis:

- **Not space efficient -** For small mathematical conjectures there is not likely to be that many hypothesis. Thus the overhead of carrying around a complete syntactic copy of every hypothesis at every node is not a real efficiency concern. However, with SPADE VCs this is certainly not the case. Even for the small examples considered so far the number of hypothesis can be quite large. In CS-note 12 the proof annotated polish flag example resulted in VC `procedure_partition_section_5` with 27 hypothesis. In real world examples, Praxis have seen this number rise well above 100.

  Note that having direct access to the hypothesis at each node avoids the need for accessing an external structure. Avoiding this extra layer of computation should lead to a faster implementation. However, as a result of not being space efficient, the extra overhead of managing larger quantities of data may have an adverse affect on the speed of the Prolog virtual machine.

- **Managing hypothesis allows for grouping -** As noted in CS-note 13 it may be advantageous to consider grouping together related hypothesis to help orient the user within a proof and constrain the proof search.

  Some initial concepts for this grouping mechanism are considered. A particular hypothesis may belong to many or no groups. For example an inequality hypothesis such as `i<j` might be in a group of hypothesis that mentions `i` and another group that mentions `j`. A trivial hypothesis such as `0=0` might not belong to any groups. This grouping mechanism can also be used to highlight hypothesis of particular significance. For example, there might be a single group containing an induction conclusion.

  Such grouping of hypothesis is likely to require an over arching structure which can refer to the available hypothesis and arrange them accordingly. The current practise of storing local copies of each hypothesis makes this problematic.

- **Managing hypothesis allows for annotated hypothesis -** There may be a desire to annotate some hypothesis to encode extra information. For example consider the following code, taken from [1] page 317:

  ```
  function Find_Smallest(Arr: Array_Type;
                         L, U: Index_Type) return Index_Type
  --# pre 1 <= L and L <= U and U <= Max_Table_Size;
  --# return The_Smallest(Arr, L, U);
  is
     K: Index_Type;
  begin
     K := L;
     for I in Index_Type range L+1 .. U loop
        if Arr(I) < Arr(K) then
           K := I;
        end if;
        --# assert 1 <= L and L+1 <= I and
        --#             I <= U and U <= Max_Table_Size and
        --#             K in Index_Type and
        --#             K = The_Smallest(Arr, L, I);
     end loop;
     return K;
  end Find_Smallest;
  ```

  This produces, amongst others, the following VC:

  ```
  For path(s) from assertion of line 32 to assertion of line 32:
  (Taking the if branch)

  function_find_smallest_3.
  ...
  H7:    k = the_smallest(arr, l, loop__1__i) .
  ...
          ->
  ...
  C7:    loop__1__i + 1 = the_smallest(arr, l, loop__1__i + 1) .
  ...
  ```

  Note that conclusion `C7` contains no reference to the variable `k`. The aliasing information that `k` is related to `loop__1__i + 1` has been lost in generating the VCs. This information might be recovered and recorded as annotations alongside hypothesis `H7`.

  Another example is reducing large hypothesis to their essential components. These lossy-hypothesis might be used during the planning phase, only resorting to the full hypothesis where necessary.

  To support these cases it is necessary to have a managed structure of hypothesis. Without this structure it would be impossible to distinguish between the real hypothesis and those with extra annotations.

The current CLAM approach is close to a suitable implementation. It is proposed to replace the CLAM structures with unique names that point to these structures. All of the hypothesis for a VC are held in a single globally accessible data structure. This will map from the hypothesis name directly to the hypothesis structure. As the hypothesis names should all be unique, this will give $O(1)$ access to their corresponding structures. For example:

```
hypothesisToStructure(HypName, HypStructure)
```

Grouping of hypothesis can be now be achieved by using the hypothesis names. A group might be held as a list of hypothesis names. For example:

```
aboutGroup(GroupName, HypNamesInGroup)
```

It will still be necessary to refer to the hypothesis during the proof plan. Two kinds of hypothesis are identified:

- **Local hypothesis -** Local hypothesis must be referenced locally from within the nodes, as they are not true in a global context. Therefore associated with each node will be a list including the names of all local hypothesis.

  Note that the hypothesis are still stored globally, only their name is local to nodes. Consider the first branch of a two way case split that introduces a new local hypothesis `hyp10`. This branch of the case split might be explored down many depths, each recording that they have access to `hyp10`. Although `hyp10` is mentioned many times, its structure is recorded once in a globally accessible location. The hypothesis `hyp10` is not mentioned anywhere in the second branch of the case split.

- **Global hypothesis -** The global hypothesis may be referred to as if for the local hypothesis. However, this is unnecessary. Repeating the names of all the global hypothesis at every node is inefficient. Further, it prevents naturally extending the available global hypothesis where globally true information, such as a separate conclusion of the same VC, is proved.

  Instead the global hypothesis may be recorded in a group. This group is then explicitly referenced in each node. Such a mechanism for group inclusion of hypothesis might also be exploited for constraining the hypothesis available. For example, instead of referring to the group of all hypothesis, a node may refer to a careful selection of focused groups instead.

There are some important disadvantages with this proposal:

- **Can not use Prolog matching directly -** Currently in CLAM the names and structure of hypothesis is available, allowing Prolog matching to be performed on these structures directly. In the proposal it will first be necessary to translate from the names to their corresponding name and structure pairs.

- **Efficiency of translation -** The translation from names to name and structure pairs may be performed often, prompting an efficiency concern. It is thought that each group of hypothesis presents a conceptually sound place to perform some caching. A list of all the name and structure pairs in a group could be calculated and cached whenever any changes are made to the group. This list could then be returned directly where requesting the structures of the group. The more groups or individual names mentioned the more expensive this process will become. The number of local hypothesis at a node are expected to small. The use of many groups is executed to be rare. Thus the caching mechanism proposed might help reduce the impact of the new structure.

2.2.1 Note on Implementation

The primary issue here is translating from names to name and structure pairs. Such an operation would need to be performed wherever direct hypothesis matching is used in CLAM. In particular, each of the methods will need to be modified, perhaps adding the translation phase as a new piece of method-language. The implementation of the global hypothesis structure store, grouping mechanism and caching facilities should not present any tricky problems.

3 Managing Proof Guiding Components

Like CLAM, the PLANNER will have a number of proof guiding components that need to be managed. These components are generally inspired as a means to naturally extend the facilities of the planner. The original proof guiding component is the method. This has expanded to include critics, sub-methods, super-methods and hints. Note that tactics are also considered as proof guiding components, due to their strong association with methods.

These proof guiding components are considered separate entities to rules, despite their bundling together in the CLAM library mechanism. Therefore the treatment of these components is strictly a PLANNER decision and not related to integration. While a new rule organisation is seen as necessary, the current library mechanism may be quite sufficient for the proof guiding components.

3.1 Methods and Critics in CLAM

The main proof guiding components of CLAM are methods and critics. It is helpful to give a brief summary of these:

3.1.1 Method

- **Specification**
  - **Pre-condition -** The conditions that must be met for the method to be applied. These are expressed using a special purpose method language.
  - **Action -** How to apply the method as part of the proof plan. These are expressed using a special purpose method language.
- **Execution**
  - **Tatic ($Tatic_1 \cdots Tatic_i$) -** A procedure capable of performing the method specification in some logical environment. It is likely that different tactics will be required for different logical environments.
- **Exception handling**
  - **Critic ($Critic_1 \cdots Critic_j$) -** A heuristic that is capable of recognising certain patterns of problems in trying to meet a methods pre-condition and propose solutions to patch these up.

3.1.2 Critic

- **Specification**
  - **Pre-condition -** The conditions that must be met for the critic to be applied. These will rely heavily on examining the partially applied method Pre-conditions. These are expressed using a special purpose method language.
- **Execution**
  - **Action -** How to apply the critic as part of the proof plan. These are expressed using a special purpose method language.

3.2 Working with Proof Guiding Components in CLAM

A number of key observations are made about the use of proof guiding components inside CLAM:

- **Proof guiding components are bundled together -** Each group of proof guiding components are bundled together in an appropriate directory in the library. For example:

```
|-Proof Guiding Components Library\
      |
      |-Methods\
      |
      |-Critics\
      |
      |-Sub-Methods\
      |
      |-Hints\
```

Note that tactics are not held in this library. The tactics are grouped together with the object level components.

- **Associations through shared names -** There are associations between different proof guiding components. For example, every method has an associated tactic and every critic has an associated method. These relationships are recorded by using the same name for related components. Interestingly, earlier versions of CLAM did explicitly record the associated tactic of a method. This was removed in later versions of CLAM by exploiting the one to one relationship between methods and tactics, giving each corresponding pair the same name.

- **Associations are bidirectional -** Given that method and tactic pairs have the same name, it is possible to find from a tactic the corresponding method and from a method the corresponding tactic. Such bidirectional associations may be useful. For example, it is necessary to find associated tactics from methods to build the specific tactic described by a proof plan. Further, where the user requests the execution of a single tactic, it may be helpful to merely present or reason with the information in the corresponding method.

- **Type through primary functor -** The type of each proof guiding component is specified as the primary functor of the component. For example:

  ```
  critic(...).
  method(...).
  hint(...).
  submethod(...).
  ```

  This type is essential in finding the appropriate proof guiding components at different stages in the proof.

3.3 Propose: Flexible Layout With Explicit Naming

A few key observations can be made about the current situation in CLAM and the relative merits of possible changes:

- **Tactics as external proof checker program interface -** Methods are not associated with the procedural content of a tactic, but with the name of the tactic, under the assumption that the tactic with this name performs some particular operation. This view stems from the observation that specific-tactics are the eventual output from a proof planner. The execution of these specific-tactics happens entirely outside of the planning system.

  Thus there is a need for tiny proof guiding components to provide stub-like information about the tactics. This might be presented as a number of components in the following form:

  ```
  tactic(Name, ObjectLevel).
  ```

  As noted later, recording the associations between proof guiding components within proof guiding components becomes untidy. Thus this tactic does not make any association with a method. It merely declares to the system that there exists a tactic with the given name in the given object-level.

- **Final structure always undecided -** The relationship between methods and critics might inspire a hierarchical structure of the form:

  ```
  |-Method\
        |
        |-Tactics\
        |
        |-Critics\
  ```

  This may be taken yet further, separating out the various components of each method. For example, isolating the pre-conditions from the actions.

  However, by their very nature the proof guiding components are subject to unpredictable change. Imposing such a structure is likely to hinder the implementation of new emerging ideas. Further, as noted above, although it is logical to think in terms of each method having its own tactic, in practise the tactics together from an interface to the proof checker that is a single discrete unit.

- **Shared names are not flexible -** The implicit naming convention used to encode the hierarchy of proof guiding components is considered inflexible as they can only represent one to one associations. For example, it is conceivable that a group of methods may all be implemented via the same (powerful) tactic. It is not possible to give all the methods the same name as the tactic. This may lead to some untidy fudges, such as making copies of the powerful tactic under different names, or introducing some dummy tactics that act as aliases.

- **Shared names are not unique -** The use of shared names means that there is not a unique name for proof guiding components. Instead, it is necessary to use type, and in case of critics associations, to determine the actual component being considered. Introducing unique names would require renaming a number of the CLAM proof guiding components.

It is proposed that the current CLAM approach of bundling each kind of proof component in their own directory should remain. Tactics are seen as proof checker components and are included in this. Note that there is no reason preventing all of the tactics from being in the same file. Thus the current library approach taken in CLAM remains the same in the PLANNER, with minor changes to the location of tactics.

The only real problem with the current situation is the recording of associations through shared names. It is proposed to introduce an explicit and more flexible facility for this. It is important that the bidirectional nature of shared names remains. Further the restriction of one to one associations should be removed.

The initial proposal was to record association information directly within the proof guiding components. This would essentially copy the behaviour of earlier versions of CLAM. Such a structure might require four attributes associated with each proof guiding component:

- **A unique name**

- **A type specifier**

- **Components being pointing at**

- **Other components pointing at this component**

The last two of these attributes are unfortunate. They requires the addition and maintenance of extra information directly inside the proof guiding components. Further, it is necessary to repeat much of the information to specify the bidirectional nature of some associations.

Instead of inserting association information directly inside the proof guiding components, it can be provided inside an external file. This file can be parsed to automatically generate a suitable data-structure that records the associations. The file can also serve to explicitly document the proof guiding components available within the system. Each proof guiding component should introduce itself in a standard form. For example:

```
wave                   is_a method.
fertilize              is_a method.
fertilize_tac          is_a tactic.
join-wave-fronts       is_a critic.
split-wave-fronts      is_a critic.
meta-ripple-position   is_a critic.
meta-ripple-direction  is_a critic.
...
```

Once introduced the proof guiding components can be loaded into the system. Note that this loading is separate to the meta-level planning decision of which proof guiding components to make available during proof planning. This is merely adding the available components into the system. The loaded proof guiding components might be held in predicates of the form:

```
proofcomponent(UniqueName, ComponentType, type(UniqueName, appropriate-structure-for-component)).
```

Initially some thought was put into trying to establish a standard notation for describing associations between proof guiding components. However, the diverse structure and unpredictable future nature of proof guiding components meant that this came across as more confusing than helpful. Instead, the associations between components can be specified however felt suitable for the kind of components. Where there is bidirectional associations the information should not be repeated. For example:

```
wave uses_tactic fertilize.
wave has_critics [join-wave-fronts,
                  split-wave-fronts,
                  meta-ripple-position,
                  meta-ripple-direction,
                  casesplit,
                  generalize,
                  lemma-calculation,
                  induction-revision,
                  lemma-speculationwave_method].
...
```

The data structures generated from parsing this file might be a number of predicates of the form:

```
association(OneUniqueName, TwoUniqueName).
```

Where bidirectional associations are being recorded the predicate should also be asserted with `OneUniqueName` and `TwoUniqueName` swapped.

3.3.1 Note on Implementation

The externals file to declare proof guiding components and their associations can be achieved using Prolog operators and the built in Prolog parser. The changes to the proof guiding components will be numerous, but trivial.

4 Planning Bookkeeping

In CS-note 20 section 3 the general organisation of planning attempts are outlined. However, this does not specify how to store necessary bookkeeping information for supporting these. The bookkeeping information records what is being planned and where, rather than the details of the planning attempts themselves. Further, this planning bookkeeping information must eventually be linked with object-level (SPADE) bookkeeping information.

4.1 Propose: Planning Bookkeeping Structure

A few points are made about the requirements of the planning bookkeeping structure:

- **Is group name and plan name -** Each plan has a name and each plan is conducted inside a named group of other plans. Every plan can be uniquely identified by its group name and plan name.
- **Group name has type -** Different groups have different types. Currently the only two types recognised are SPADE VCs and lemmas. It is not considered to be necessary to record this type. However, it may bring some slight efficiency gains in exporting proofs to SPADE as the type can be more readily identified. Further, the information should be helpful in presenting the planning actions to the user.
- **Names are not SPADE names -** Where dealing with spade VCs the names of planning groups and their plans are not necessarily the corresponding ones in SPADE. It is preferred to record this object-level information outside of the planner bookkeeper.

A straight forward collection of asserted predicates can hold the required information. The following is proposed:

```
allGroups(GroupName, TypeGroup).
allPlansInGroups(GroupName, PlanName).
```

4.1.1 Note on Implementation

The bookkeeping for planning can be achieved by recording the required information as asserted predicates.

4.2 Propose: A Conversion Structure

The planning bookkeeping structure is augmented with additional object-level information:

- **Export for SPADE VCs -** The SPADE VC plans will already have corresponding names in SPADE. Each `GroupName` will have a corresponding VC name and each `PlanName` will have a corresponding conclusion number.
- **Export for lemmas -** The lemma plans are new, and do have corresponding names in SPADE. Ideally, lemma plans should be executable inside SPADE. The intention is to create a SPADE problem (VCG and FDL file) for each lemma and then execute the plan to demonstrate the correctness of the lemma. As lemmas are initialised in the PLANNER their bookkeeping information will be added, additionally suitable object-level names should be generated and stored here.

The information required can be held in a number on asserted predicates:

```
groupToSpade(GroupName, SpadeGroupName).
planToSpade(PlanName, SpadePlanName).
```

4.2.1 Note on Implementation

Further predicates are asserted to record object-level information within the planner. Many of these will be recorded as proof information is imported from SPADE, with the remaining generated during the planing phase. The information need only be accessed as plans are exported to SPADE.

5 Information Partitioning in the Planning Space

There is a lot of important information used during planning that is not as directly visible as rules or proof guiding components. Further, where this information does not apply globally to all plans, it must be suitably partitioned.

5.1 Information Partitioning in CLAM

Important points about information partitioning in CLAM:

- **Naming nodes -** As discussed in CS-note 5 each node of the planning attempt contains `PlanName`, the name of the conjecture being planned:

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

  This alone is sufficient to partition the planning attempts for CLAM.

- **Other information across all plans -** Other than the partitioning seen in nodes, CLAM does not partition any other planning information. Instead the information is global and applies to all plans simultaneously. For example, the availability and ordering of proof guiding components is the same for all plans.

- **Information available as asserted -** The proof guiding components available are those that have been loaded into the system. CLAM never unloads any proof guiding components. Thus once loaded a proof guiding component becomes globally available to every plan.

- **Application order as loading order -** The order in which proof guiding components are attempted is determined by the order in which they are loaded. The earlier the component is loaded the earlier it will be used. For example, the waterfall of methods described in CS-note 5 is loaded into the system via:

  ```
  load_ind_plan(1):-
          delete_methods,delete_submethods,
          lib_load(mthd(elementary/1)),
          lib_load(mthd(normal/1)),
          lib_load(mthd(equal/2)),
          lib_load(mthd(eval_def/2)),
          lib_load(mthd(fertilize/1)),
          lib_load(mthd(wave/2)),
          lib_load(mthd(generalise/2)),
          lib_load(mthd(induction/2)),
          list_methods.
  ```

- **Current position through backtracking -** The application order of proof guiding components is achieved by relying on Prolog backtracking. Recursion is used to reset the current position. In moving to a blackboard architecture it will no longer be possible to directly rely on such low level Prolog behaviour. This will be particularly pertinent where meta-planning can introduce new proof guiding components or even rearrange their application order.

5.2 Propose: Use Bookkeeping Partitioning

The bookkeeping information identified how the planning space will be partitioned. Together `PlanGroup` and `PlanName` are sufficient to suitably constrain the scope of all information. For example, the node predicate will become:

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

5.2.1 Note on Implementation

All information will be global, local to a group of plans or local to a particular plan. This partitioning can be achieved by adding `PlanGroup` and `PlanName` to the information predicates.

5.3 Propose: Record Information Needed

Additional information required should be recorded explicitly and suitably partitioned. The availability and application order for proof guiding components might be specified using:

```
order(PlanGroup, PlanName, UniqueName, ComponentType, OrderNumber).
```

The current position in this ordering will be stored with the nodes:

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

5.3.1 Note on Implementation

The implicit information in CLAM can generally be made explicit in the PLANNER without particular difficulties. Note that this is continuing the same ideas that resulted in adding the `node` predicate into the critics version of CLAM.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.