Bill J. Ellis

June 9, 2005

Revisiting the SPADE Proof Checker

Contents

[*]Hums are short notes intended for distribution between those involved with EPSRC grant GR/T12289/01. Hums describe ϵ-baked ideas, where $1 \geq \epsilon \geq 0$. (Hum refers to both the Praxis Humming bird and Winnie-the-Pooh's indirect approach to writing: "Poetry and Hums aren't things which you get, they're things which get you.").

1 Introduction

Hum 4 describes the changes made to the SPADE Proof Checker to create SPADE Proof Checker-Tame. The changes were strongly inspired by the version of the SPADE Proof Checker used within NuSPADE, with an improved, configurable, implementation. As noted in Hum 4, these changes would need to be fine-tuned before practical use. Here, these changes are revisited with the imminent task of translating discovered plans to SPADE Proof Checker command logs.

2 Observations

2.1 Robust command logs and why not

For a short while following Hum 4 there was tentative plans for more extensive modifications to the SPADE Proof Checker than those seen in SPADE Proof Checker-Tame. In particular, the practise of referencing hypotheses and conclusions by numbers was proposed to be replaced by simply referencing the actual term structures of the hypotheses and conclusions. This is a common approach taken in theorem provers significantly increasing the repeatability and understandability of proof scripts. Such changes to the SPADE Proof Checker would be desirable in general. Further, they would slightly ease the translation process from plans to SPADE Proof Checker command logs. However, it became clear that these changes would require a few significant changes to the SPADE Proof Checker. This should probably be undertaken as a genuine SPADE Proof Checker enhancement rather than as a side effect for improved control.

Note, that, as a consequence of deeper changes to the SPADE Proof Checker, the command logs produced by the SPADEase-Planner tend not to refer to explicit hypothesis or conclusion numbers creating much more robust scripts.

2.2 Improve rewrite commands in the SPADE Proof Checker

Nearly every proof planning step can be expressed as a simple rewrite. It is estimated that about 80% of the proof plan steps can be directly translated into the following:

- **rewrite**
  - **implication rule**
    - **applied to hypotheses**
    - **applied to conclusion**
  - **equality rule**
    - **applied to hypotheses**
    - **applied to conclusion**

Currently all such steps are indeed achieved through a small number of general rewrite functions. However, these translate into ghastly recursive nightmares of proof commands. In practise, this renders the command logs essentially unreadable - eroding their value and making debugging a frightening process. Worse, these translations are not even accurate. The introduction and instantiation of variables implicitly rely upon the current ordering of hypotheses and the detailed implementation of the `infer` or `replace` command. This problem is noted, but not addressed, in Hum 4 as the need for a new `infer_tame` command. Yet further, the SPADE Proof Checker commands do not automatically support the full power of rewrites. The ability to perform a rewrite is dependent on the presentation of the supporting rule (rather than its fundamental logic). Correctly handling these cases would require making the planner aware of both the logic of rules and their presentation within the SPADE Proof Checker. Maintaining and using this additional information further increases the complexity of the planning and translation process[1].

It is argued that these four rewrites could be directly built into the SPADE Proof Checker. Although this would involve writing new code, it should be possible to build upon the existing SPADE Proof Checker predicates. Arguably, a central implementation of these key operations (rather than a collection of small amendments to existing commands) is a more coherent policy.

[1]The original planner simply avoided the problem by not using rules to their logical potential, instead using rules in the manner the SPADE Proof Checker would expect.

2.3 Rewrite revisited

Unfortunately, it is restrictively complex to build upon the existing implementations of `replace` and `infer`. These commands and their effects are fundamentally intertwined with user interaction and a complex collection of database accesses. This presents the dilemma:

- **Keep original version -** The original version made as few changes to the SPADE Proof Checker as possible. This is flawed for rewrites that introduce new term structure. This complicates the use of rules (require different translations depending on the spade rule presentation). The translations themselves are particularly difficult to comprehend. And, changes must still made to the SPADE Proof Checker. Planning for a pure SPADE Proof Checker would be exceptionally difficult.

- **Accept new commands -** The SPADE Proof Checker supports clean, powerful, neatly encapsulated commands for the key rewrite steps. The translations become coherent, and all the fiddly (and broken) aspects of translation are eliminated. However, is the SPADE Proof Checker still the same system?

The former case involves a fair amount of hassle to provide an implementation the correctness of which is no more guaranteed than the latter. In both cases the SPADE Proof Checker remains a separate component, a proof checker. It is accepted, quite quickly, that the later case is not ideal. However, this is a practical solution to the problems found in automatically controlling the SPADE Proof Checker. It could be done another way, but the effort involved would be far too costly.

2.4 Just the key features

Just the key features for regular planning will be considered. More sophisticated planning features, such as generalisation or induction, will be postponed for now.

3 One new SPADE Proof Checker command

A single, but very significant, new command was added to the SPADE Proof Checker: `tame_rewrite`. This command supports the rewriting of hypotheses and conclusions based on different properties:

- **using rule -** The full logic of every rule in the SPADE Proof Checker may be used to perform a rewrite.

  ```
  tame_rewrite HYPCONC : ORIGINAL : POSITION with LHS to RHS if CONDITION using RULE in DIRECTION
  ```

- **using hypothesis -** Every hypothesis that can be presented as a rule may be used to perform a rewrite. For example, the hypothesis $i + i = j$ may be used to replace $i + i$ with j.

  ```
  tame_rewrite HYPCONC : ORIGINAL : POSITION with LHS to RHS if CONDITION from FORMULA in DIRECTION
  ```

- **using evaluate -** Every term structure that can be evaluated can rewritten with its evaluated form. For example, the term $1 + 1$ may be replaced with 2.

  ```
  tame_rewrite HYPCONC : ORIGINAL: POSITION with EXPRESSION is VALUE
  ```

- **using hypothesis as true -** Every hypothesis is true. Thus every hypothesis A can be thought of as the equality $A = true$. Terms may be rewritten (to true only) using this property.

  ```
  tame_rewrite HYPCONC : ORIGINAL: POSITION where HYP
  ```

The command `tame_rewrite` is invoked with complete details of the operation to be performed. Technically, this involves repeating some information in some cases. However, as these commands are not intended for interactive use, the additional information, and consequent double checking of information, is beneficial and imposes no problems. The various arguments seen are:

- **HYPCONC -** Is either `hyp` or `conc` to denote whether a hypothesis or conclusion is being rewritten.

- **ORIGINAL -** The full term structure of the hypothesis or conclusion being processed.

- **POSITION -** The position of the subterm within the full term structure being rewritten. The position is identified by a (reversed) list of integers that describes a path through the tree structure of the full hypothesis or conclusion.

- **LHS -** The left hand side being rewritten. This should match the subterm at $POSITION$ in $ORIGINAL$.

- **RHS** - The expected new subterm following the rewrite.
- **CONDITION** - Any condition to be discharged to support the application of a rule. In most cases no condition is required, so this is just true. In other cases this will contain a term that must match exactly with one of the existing hypotheses.
- **RULE** - The name of a rule.
- **DIRECTION** - Is either `lefttoright` or `righttoleft` depending on which way round **RULE** is being employed.
- **FORMULA** - A rewrite formula that currently exists in the hypotheses. This need not match exactly with a hypothesis, as conditional formulae may be used ($CONDITION- > FORMULA$).
- **EXPRESSION** - The left hand side being evaluated. This should match the subterm at $POSITION$ in $ORIGINAL$.
- **VALUE** - The single integer value that results from evaluating $EXPRESSION$.
- **HYP** - The full term structure of a hypothesis that is being used to rewrite a term structure to true.

3.1 Limitations and known issues

There are a few limitations and known issues for the implementation of this powerful rewrite command.

- **Polarity** - The legal direction for the application of rules based on implication depends on the polarity of the subterm being rewritten. To avoid dealing with this problem, implication rules may only be applied to subterms that are the whole term or a top level conjunct of the whole term. Thus, the polarity of the subterm being rewritten is always the same as the polarity of the top level term. The polarity of the top level term can be immediately determined based on whether this is a hypothesis or a conclusion. Note that this restriction only applies to rewriting with implication based rules.
- **Quantifiers** - Rewrite rules can legitimately rewrite quantified expressions, and this remains supported. However, rewriting within quantified expressions introduces implementation problems as it requires working with variables that are visible in a constrained scope and requires preventing the user from rewriting the type specification part of the qualifier. In practise, rewriting within quantifiers is valuable, but not required. Thus rewriting within quantified expressions is not supported.
- **Array access** - In the expressions `element(A,I)` and `update(A, I, B)` `I` is a list of array index accesses of the form `[I`^{1}`, ..., I`^{n}`]`. While the elements of this list may be rewritten, rewriting the list itself would cause problems. Such a bad rewrite is not explicitly prevented. However, there should not exist rules that manipulate whole lists as this would violate the FDL type model.

  In general, any term which represents a function is likely to have additional constraints over regular terms. These cases warrant particular attention. Fundamentally, the subterm indexing facility behind the `tame_rewrite` command should be more concerned about the semantics of the terms being deconstructed than the current implementation which primarily considers the syntax of terms.
- **Soundness concerns** - This rewrite command is too powerful. It should be a tactic, in the HOL style, that is expressed concisely in terms of a series of clearly sound low level operations. Implementing such low level operations in the SPADE Proof Checker, creating a mechanism to express combinations of these, and writing the suitable tactic is entirely feasible (see HOL) but restrictively time consuming. Hence the unfortunate existing implementation.

  Note that there is no know soundness weaknesses in the existing implementation. However, it is known that there are some unknowns as a detailed study of the FDL type model and more esoteric features of the SPADE Proof Checker[2] were not pursued. There may even be some unknown unknowns. This code does not meet the SPADEase goal of being industrially viable - a strong implementation would address these unknowns and follow the HOL model outlined above.

[2] Such as rules with conditions that are executable Prolog goals...

4 Executive Summary

An industrial strength version of a Proof Planner requires a secure coupling to an external proof checker. In its original form, it is effectively impossible to achieve this with the SPADE Proof Checker. Following the first draft of refinements, it is conceivably possible to achieve such coupling with the SPADE Proof Checker, however this is rather complex. Thus, the plan is to drop a key extra command into the SPADE Proof Checker, further diluting its original form, raising soundness concerns, yet dramatically easing coupling to a Proof Planner.