Refactoring Proofs with Tactician

Mark Adams[1,2](✉)

[1] Proof Technologies Ltd., Worcester, UK
Mark@proof-technologies.com
[2] Radboud University, Nijmegen, The Netherlands

Abstract. Tactician is a tool for refactoring tactic proof scripts for the HOL Light theorem prover. Its core operations are packaging up a series of tactic steps into a compact proof with tactical connectives, and the reverse operation of unravelling compact proofs into interactive steps. This can be useful for novices learning from legacy proof scripts, as well as for experienced users maintaining their proofs. In this paper, we give an overview of Tactician's core capabilities and provide insight into how it is implemented.

1 Introduction

Although now over 30 years old, Paulson's subgoal package [9] is still the predominant mode of interactive proof in various contemporary theorem provers, including Coq [4], HOL4 [10], HOL Light [6] and ProofPower [2], and is still used by some in the Isabelle [11] community. In recent years it was used extensively in the verification of the seL4 operating system microkernel [7] in Isabelle/HOL [8], and in the Flyspeck mathematics formalisation project [5] in HOL Light, for example.

Despite its widespread use, user facilities remain basic and lack useful extended features. One desirable facility is automated proof refactoring, for transforming a proof script into a more suitable form, where "more suitable" depends on the user's needs, and might mean more compact, more efficient, more readable, more maintainable or easier to step through interactively. The understandability of various large proof scripts from Flyspeck, for example, would improve greatly if their tactic proofs could be cleaned up to be more coherent and easier to step through.

In this paper, we describe Tactician, a tool for refactoring HOL Light tactic proof scripts, explaining how key parts of its implementation work. In Sect. 2, we provide the motivation for automated proof refactoring. In Sect. 3, we show examples of Tactician's two main refactoring operations. In Sect. 4, we explain how Tactician refactors proofs. In Sect. 5, we explain how it captures tactic proofs in state. In Sect. 6, we report on experiences and limitations. In Sect. 7, we present our conclusions. This paper extends an earlier workshop paper [1], by explaining the proof refactoring mechanism, filling out more detail about how tactic proofs are captured, and covering experiences and limitations.

D. Bianculli et al. (Eds.): SEFM 2015 Workshops, LNCS 9509, pp. 53–67, 2015.
DOI: 10.1007/978-3-662-49224-6_6

Tactician is open source and can be downloaded from [12]. We explain how it works with extracts from its implementation in the OCaml dialect of ML, although these extracts are simplified for illustrative purposes.

2 Background and Motivation

The subgoal package is simple in concept and yet remarkably effective in practice. The user starts with a single main goal to prove. They then apply a series of tactic steps to break down the goal into hopefully simpler-to-prove subgoals. Each tactic application results in one or more new subgoals, or otherwise completes its subgoal, in which case focus shifts to the next subgoal. The proof is complete when each leaf in the tree of subgoals has been proved.

Behind the scenes, the subgoal package is keeping everything organised by maintaining a proof state that includes a list of remaining subgoals and a justification function for constructing the formal proof of the main goal from the formal proofs of the remaining subgoals. Tactics are implemented as functions that take a goal and return a subgoal list plus a justification function. The proof state is updated every time a tactic is applied, incorporating the tactic's resulting subgoals and justification function. Once the proof is complete, a theorem stating the main goal can be extracted using the final justification function.

```
g `!f:A->B. (!y. ?x. f x = y) <=> (!P. (?x. P(f x)) <=> (?y. P y))`;;
e (GEN_TAC);;
e (EQ_TAC);;
e (MESON_TAC []);;
e (DISCH_THEN (MP_TAC o SPEC `\y:B. !x:A. ~(f x = y)`));;
e (MESON_TAC []);;
top_thm ();;
```

Fig. 1. An interactive "g and e" style proof of HOL Light's SURJECTIVE_EXISTS_THM.

The user writes their proof script as a series of commands in the ML programming language. The first draft is typically written in the interactive proof style, which starts with the command to set the proof goal (called g in HOL Light), proceeds with a potentially long series of tactic steps each using the tactic application command (called e in HOL Light), and often ends with a command to extract the theorem result (called top_thm in HOL Light), as illustrated in Fig. 1. This version is usually not beautiful, but does the job of proving the theorem. For clarity, the user may have inserted indentations and/or ML comment annotations to make explicit where the proof branches, but if not the proof script does not convey any information about the structure of the proof. Worse still, the proof steps may be interspersed with other, unrelated ML commands in the script, further obscuring the proof.

What often happens next is that the proof script is cleaned up, or *refactored*, to become more succinct. This refactoring will typically involve packaging-up the tactic steps into a single, compound tactic, and then using the batch proof command (called `prove` in HOL Light) for performing the completed proof and extracting the theorem result, as illustrated in Fig. 2. If done well, the resulting compound tactic will be neater and more concise because it can factor out repeated use of tactics, for example when the same tactic is applied to each subgoal of a given goal.

```
prove
 (‘!f:A->B. (!y. ?x. f x = y) <=> (!P. (?x. P(f x)) <=> (?y. P y))‘,
  GEN_TAC THEN
  EQ_TAC THENL
   [ALL_TAC; DISCH_THEN (MP_TAC o SPEC ‘\y:B. !x:A. ~(f x = y)‘)] THEN
  MESON_TAC []);;
```

Fig. 2. A batch “prove” style proof of SURJECTIVE_EXISTS_THM.

Tactic connectives are used to package up steps into compound tactics. The binary THEN connective applies its right-hand side (RHS) tactic to all of the subgoals resulting from its left-hand side (LHS) tactic. The binary THENL differs in that its RHS argument is a list of tactics rather than a single tactic, where the nth tactic in this list gets applied to the nth subgoal resulting from the LHS tactic. The identity tactic ALL_TAC makes no change to the goal, which can be useful in the RHS list of a THENL to make a branch skip the list.

These packaged-up proofs feature heavily in the source code building up the standard theory library of the HOL4 and HOL Light systems. They were also used in Flyspeck for a few years, because they bring the proof together as a single ML statement, making it easier to manage.

In light of this, one can see that a capability for automatically refactoring tactic proof scripts between the interactive “g and e” style and the packaged-up “prove” style would be valuable to the user, for three reasons:

Tidying. Having created a new interactive proof, the process of tidying it and packaging it up into a batch proof can be long and tedious, and for proofs that run into dozens of lines it can be easy to miss opportunities to make the proof more concise or readable. Doing this automatically could both save effort and result in better proofs;

Learning. Most of the best examples of tactic proofs are stored as packaged-up batch proofs. Novice users currently have to laboriously unravel these into interactive proofs if they want to step through these masterpieces and learn how the experts prove their theorems. This can be even more tedious than packaging a proof up, because the user does not know which tactics apply to more than one subgoal and thus need to occur more than once in the unpackaged proof. Automation would improve access to the wealth of experience that is held in legacy proof scripts;

```
prove
 (‘!x. &0 < x ==> &0 < inv(x)‘,
  GEN_TAC THEN
  REPEAT_TCL DISJ_CASES_THEN
       ASSUME_TAC (SPEC ‘inv(x)‘ REAL_LT_NEGTOTAL) THEN
  ASM_REWRITE_TAC[] THENL
   [RULE_ASSUM_TAC(REWRITE_RULE[REAL_INV_EQ_0]) THEN ASM_REWRITE_TAC[];
    DISCH_TAC THEN
    SUBGOAL_THEN ‘&0 < --(inv x) * x‘ MP_TAC THENL
     [MATCH_MP_TAC REAL_LT_MUL THEN ASM_REWRITE_TAC[];
      REWRITE_TAC[REAL_MUL_LNEG]]] THEN
  SUBGOAL_THEN ‘inv(x) * x = &1‘ SUBST1_TAC THENL
   [MATCH_MP_TAC REAL_MUL_LINV THEN
    UNDISCH_TAC ‘&0 < x‘ THEN REAL_ARITH_TAC;
    REWRITE_TAC [REAL_LT_RNEG; REAL_ADD_LID; REAL_OF_NUM_LT; ARITH]]);;
```

Fig. 3. The original batch proof of REAL_LT_INV from HOL Light's `real.ml`.

Maintenance. Proofs need to be maintained over time, due to changes in the theory context in which the theorems are proved. If the proofs are packaged up, then they will need to be unpackaged, debugged and then repackaged, which again would be considerably easier with automated support.

3 Usage

Tactician can be loaded into a HOL Light session at any stage, and from that point onwards silently captures proofs as they are executed, outputting refactored versions upon request. We illustrate it usage here with REAL_LT_INV, proved in the HOL Light standard theory library (see Fig. 3).

3.1 Proof Unravelling

Trying to manually unpick the original packaged proof of REAL_LT_INV is an arduous task. From looking at the text of the script, use of THENL reveals that the proof branches at various points, but it is not clear exactly where these branches start, which tactics get applied to more than one branch, and which branches are finished off within the RHS of the THENL connectives rather than carry on further into the proof script.

Using Tactician, the proof can be automatically unravelled into "g and e" style, with branch points optionally annotated to reveal its true structure (see Fig. 4). We can see that the application of REPEAT_TCL resulted in three subgoals, with the first finishing in the first branch of the first THENL, the second finished off by ASM_REWRITE_TAC prior to the first THENL, and the third continuing beyond the second branch of the first THENL to split and continue to the end of the packaged proof. Once expressed in "g and e" style, the proof is ready to be replayed interactively by stepping through individual tactic applications.

```
g `!x. &0 < x ==> &0 < inv(x)`;;
e (GEN_TAC);;
e (REPEAT_TCL DISJ_CASES_THEN
        ASSUME_TAC (SPEC `inv x` REAL_LT_NEGTOTAL));;
(* *** Branch 1 *** *)
e (ASM_REWRITE_TAC []);;
e (RULE_ASSUM_TAC (REWRITE_RULE [REAL_INV_EQ_0]));;
e (ASM_REWRITE_TAC []);;
(* *** Branch 2 *** *)
e (ASM_REWRITE_TAC []);;
(* *** Branch 3 *** *)
e (ASM_REWRITE_TAC []);;
e (DISCH_TAC);;
e (SUBGOAL_THEN `&0 < --inv x * x` MP_TAC);;
(* *** Branch 3.1 *** *)
e (MATCH_MP_TAC REAL_LT_MUL);;
e (ASM_REWRITE_TAC []);;
(* *** Branch 3.2 *** *)
e (REWRITE_TAC [REAL_MUL_LNEG]);;
e (SUBGOAL_THEN `inv x * x = &1` SUBST1_TAC);;
(* *** Branch 3.2.1 *** *)
e (MATCH_MP_TAC REAL_MUL_LINV);;
e (UNDISCH_TAC `&0 < x`);;
e (CONV_TAC REAL_ARITH);;
(* *** Branch 3.2.2 *** *)
e (REWRITE_TAC [REAL_LT_RNEG;REAL_ADD_LID;REAL_OF_NUM_LT;ARITH]);;
```

Fig. 4. An interactive version of REAL_LT_INV.

3.2 Proof Packaging

Tactician can automatically package up a "g and e" style proof into a batch proof, identifying opportunities to factor out repeated use of tactics to make the batch proof more concise. We concentrate here on how it can do a better job at making the batch proof readable than what has been done manually in the HOL Light standard theory library.

A good technique to be used in packaging up a tactic proof is to make the main branch of the proof clear by separating it out from minor branches that get discharged in relatively few steps. This is done by completing any minor branches within the RHS of a THENL connective as soon as they arise in the proof, but keeping the proof of the main branch outside of this by performing a null step in its corresponding place in the THENL RHS by use of the identity tactic ALL_TAC. Thus all subsequent steps after the THENL RHS are concerned only with the main branch, and the THENL RHS is only concerned with the minor branches.

This technique is used throughout the HOL Light standard theory library, although various opportunities to use it have been missed, as is the case with

```
prove
 (`!x. &0 < x ==> &0 < inv(x)`,
  GEN_TAC THEN
  REPEAT_TCL DISJ_CASES_THEN
       ASSUME_TAC (SPEC `inv(x)` REAL_LT_NEGTOTAL) THEN
  ASM_REWRITE_TAC [] THENL
   [RULE_ASSUM_TAC (REWRITE_RULE [REAL_INV_EQ_0]) THEN
    ASM_REWRITE_TAC [];
    ALL_TAC] THEN
  DISCH_TAC THEN SUBGOAL_THEN `&0 < --(inv x) * x` MP_TAC THENL
   [MATCH_MP_TAC REAL_LT_MUL THEN ASM_REWRITE_TAC []; ALL_TAC] THEN
  REWRITE_TAC [REAL_MUL_LNEG] THEN
  SUBGOAL_THEN `inv(x) * x = &1` SUBST1_TAC THENL
   [MATCH_MP_TAC REAL_MUL_LINV THEN UNDISCH_TAC `&0 < x` THEN
    REAL_ARITH_TAC;
    REWRITE_TAC [REAL_LT_RNEG; REAL_ADD_LID; REAL_OF_NUM_LT; ARITH]]);;
```

Fig. 5. A repackaged version of REAL_LT_INV, making use of ALL_TAC.

REAL_LT_INV. We can see from its unravelled proof in Fig. 4 that Branch 3 and subsequently Branch 3.2 represent the main branch of the proof, although from Fig. 3 we can see that these branches are partly worked on inside the RHSs of THENL connectives and partly worked on outside.

As can be seen in Fig. 5, when using Tactician to package up the unravelled proof, the two opportunities to separate out the main branch are not missed, as is evident by the two occurrences of ALL_TAC. Thus the packaged proof is easier to follow, as well as keeping a better shape.

3.3 Output

The most common usage of Tactician is to output to screen an individual refactored proof. However, there are various extended features to support the management of large proof projects. It is capable of bulk processing a file of proof scripts and exporting refactored versions of each to disk. There are also commands for exporting the graph of a proof to help visualise its goal tree, for presenting statistics about each proof including size and usage metrics, and for exporting a graph showing the dependencies between theorems.

4 Implementing Proof Refactoring

We now explain the mechanism by which tactic proofs are refactored. This is based around combining basic transformation operations performed on an abstract representation of a tactic proof called a *hiproof*.

4.1 Hiproofs

In order for a proof script to be refactored, it should first be represented in a suitable form that holds all necessary information about its structure, so that it can be correctly transformed, and sufficient information about its components, so that the result can be outputted as a standalone proof script that can be executed as a substitute for the original.

Hiproofs (see [3]) are an algebraic representation of tactic proofs that we find are at just the right level of abstraction for our purposes. They represent a proof in terms of the goal structure resulting from the execution of the proof, with each goal represented by the tactic applied to it. There is an atomic class for a tactic application on an individual goal, an identity hiproof for a null tactic application, an empty hiproof for a completed subgoal, a binary sequential composition operator written as semicolon, a binary tensor operator $\otimes$ for grouping hiproofs in parallel (for goals with multiple subgoals) and a labelling operator. The grammar is given by:

$$h ::= \mathit{tactic} \mid \mathit{id} \mid \mathit{empty} \mid h; h \mid h \otimes h \mid [\mathit{label}]\ h$$

Our representation differs from the algebra in [3] to better fit implementation of proof refactoring. There is an additional atomic class for an active subgoal, to cater for incomplete proofs. Another difference is that our tensor operator groups a list of hiproofs rather than a pair, enabling the empty hiproof to be replaced with a tensor of length 0. Note that we write our hiproof notation differently, so that tensor is written with angled brackets and comma separators.

$$h ::= \mathit{tactic} \mid \mathit{active} \mid \mathit{id} \mid h; h \mid \langle h, h, ..\rangle \mid [\mathit{label}]\ h$$

In Tactician, the labelling construct is used differently from its original intended use, where the label is supposed to represent an abstract view of a subproof that can be "zoomed in" to show the detail of the hiproof inside if desired. Instead we use labelling primarily to capture usage of the `THEN` and `THENL` tactic connectives (which both represent sequential composition).

In Fig. 6, we show a toy example of a packaged proof script and its hiproof representation. Note that, unlike the proof script, the hiproof conveys the full structure of the proof, explicitly showing that `REFL_TAC` gets applied to the two subgoals resulting from `CONJ_TAC`. For illustrative purposes, we have kept the parentheses around compound sequential composition. Note that these brackets accumulate on the left in packaged proofs, because the `THEN` and `THENL` connectives are left-associative, because they are normally used to apply their RHS argument to all of the remaining subgoals in the proof.

In Fig. 7, we show an interactive version of the toy example. The hiproof has no labels since tactic connectives are not used. Note that the sequential composition brackets accumulate on the right in "g and e" style proofs, because the LHS argument reflects a single goal in interactive proof, and the RHS argument gets applied just to its subgoals.

Each hiproof has an input and output arity, corresponding to the number of input and output goals of the hiproof respectively. To be well-formed, the

```
prove (‘!x. x = x /\ x = x‘,
       GEN_TAC THEN CONJ_TAC THEN REFL_TAC‘);;

[Label THEN]
  ([Label THEN] (Tactic GEN_TAC; Tactic CONJ_TAC);
   <Tactic REFL_TAC, Tactic REFL_TAC>)
```

Fig. 6. A toy example of a packaged proof, together with its hiproof.

```
g ‘!x. x = x /\ x = x‘;;
e (GEN_TAC);;
e (CONJ_TAC);;
e (REFL_TAC);;
e (REFL_TAC);;

Tactic GEN_TAC; (Tactic CONJ_TAC; <Tactic REFL_TAC, Tactic REFL_TAC>)
```

Fig. 7. An interactive version of the toy proof, together with its hiproof.

output arity of the LHS of a sequential composition must equal the input arity of the RHS. The arity of a hiproof can be calculated from its atoms. Tactic application has input arity 1 and output arity depending on the application. An active subgoal has input arity 1 and undefined output arity. The identity hiproof has input and output arity 1. Sequential composition has input arity of the LHS hiproof and output arity of the RHS hiproof. Tensor has input and output arity of the sum of its hiproofs' input and output arities respectively. And a labelled hiproof has the input and output arity of its hiproof.

4.2 Hiproof Transformations

The insight into the different way in which the brackets accumulate between packaged and interactive proofs provides the basis for the refactoring operations. To convert between one style and the other, the brackets need to be shifted from one side to the other. In reality, the refactoring operations have to do much more than simply shift brackets, and are compositions are various primitive refactoring transformations, but the shifting of brackets is the most fundamental aspect.

Unravelling a Hiproof. To refactor into an unravelled proof, it is first necessary to remove any labels from the hiproof, that capture the use of `THEN` and `THENL` connectives, that would be present if the original proof were a packaged proof. Then it is possible to recursively apply the associativity rule for sequential composition in the left-to-right direction (1), to group the sequential composition brackets on the right.

$$(h1; h2); h3 \longrightarrow h1; (h2; h3) \tag{1}$$

However, the ultimate aim is to output an interactive proof, which cannot involve tensors as the LHS of a sequential composition. Thus part of the right-grouping operation is to remove any such tensors by distributing them through their sequential composition RHS (2). This distribution must preserve the well-formedness of the hiproof, and the components of the resulting tensor must be lined-up so that arities match (thus $h3a$ below is not necessarily simply $h1a; h2a$).

$$\langle h1a, h1b, ..\rangle ; \langle h2a, h2b, ..\rangle \longrightarrow \langle h3a, h3b, ..\rangle \tag{2}$$

The hiproof is then ready to be printed out as ML, with the option of inserting a comment for every branching point to show the structure of the proof.

Packaging-Up a Hiproof. To refactor into a packaged proof, we first remove any labels, since the original proof might also be a packaged proof. The next task is to compact the hiproof by spotting opportunities for performing the same tactic in parallel to each component of a tensor, enabling `THEN` to be used in place of `THENL` and thus making the proof more concise. These common tactics can be at the start or the end of tensors, corresponding to the initial few tactics after a proof branches or the final few tactics at the end of each of a branch's subbranches respectively. Thus we first group sequential composition brackets on the right and compare initial segments of a tensor's hiproofs, separating out any common initial segments. We then do the same for common final segments of a tensor's hiproofs by left-grouping sequential composition brackets, using the associativity rule in the right-to-left direction (3).

$$(h1; h2); h3 \longleftarrow h1; (h2; h3) \tag{3}$$

Next, if there is a branch in a tensor that is much longer than its sibling branches, then it is extracted out of the tensor using the `ALL_TAC` technique explained in Sect. 3.2. The heuristic we use for identifying a "much longer" branch is that the branch has size of at least five, and this size is at least three times the maximum size of its siblings, where size is calculated as the number of ML atoms in its ML text. Experience has shown that this heuristic delivers good results. Having applied the heuristic, the hiproof has been optimised with separated tensors for parallel application of the same tactic. Each sequential composition is then labelled with `THENL` unless its RHS has input arity 1 or if the RHS is a tensor with the same tactic as each component, in which case it is labelled as `THEN`. The packaged hiproof is then ready to be printed out as ML.

5 Implementing Tactic Recording

In the last section, we explained how a tactic proof represented as a hiproof is refactored in Tactician. However, this assumes that the proof has been appropriately captured in program state. In this section, we explain how Tactician captures proofs, which accounts for the bulk of its implementation. First we briefly discuss the requirements for capturing tactic proofs in a suitable form.

5.1 Requirements for Capturing Tactic Proofs

It is not particularly important that the original text of the proof can be recreated in every last detail, including those parts that are redundant or that don't get executed. For our purposes, we are more interested in what does get executed, and what happens when it gets executed, which tells us more about which proof refactorings would be valid. Thus capturing the proof by a static syntactic transformation of the original proof script would not suit our purposes. Rather, the proof needs to somehow be dynamically recorded as it is executed, to capture what is actually used. We call this *tactic proof recording.*

Note that the subgoal package already dynamically captures interactive tactic proofs, simply as a list of subgoals (or actually a stack of such lists, so that interactive steps can be undone if required). However, this form is not suitable for our purposes because it does not explicitly capture the structure of the tree of goals, and neither does it carry the names of tactics used or their arguments, which we require in order to output a proof script.

We identify seven main requirements for our tactic recording mechanism:

1. To fully capture all the information needed to recreate a proof script;
2. To capture the parts of the proof script that actually get used;
3. To capture the information in a form that suitably reflects the full structure of the original proof, including the structure of the goal tree and hierarchy corresponding to the explicit use of tactic connectives in the proof script;
4. To capture information at a level that is meaningful to the user, i.e. with atoms corresponding to the ML binding names for the tactics and other objects mentioned in the proof script;
5. To be capable of capturing both complete and incomplete proofs;
6. To work both for both interactive and packaged-up proofs;
7. To work for legacy proofs, without requiring modification to the original proof script.

5.2 The Basic Recording Mechanism

Our recording mechanism is designed to meet the above requirements. It maintains a proof tree in program state, in parallel with the subgoal package's normal state. The proof tree has nodes corresponding to the initial and intermediate goals in the proof, and branches from each node corresponding to the goals' subgoals, reflecting the structure of the executed proof. Each node carries information about its goal, including a statement of the goal, an abstract representation of the ML text of the tactic that got applied to the goal, and a unique goal identity number. Active subgoals are labelled as such in place of the tactic's abstract ML text, thus enabling incomplete proofs to be represented. The tree gets added to as interactive tactic steps are executed, and deleted from if steps are undone. A hiproof that captures the proof's structure and the ML text of the tactic applied in each goal can thus be dumped from the proof tree at any stage, from which the proof can be refactored and printed (see Sect. 4).

The crucial means by which the stored proof tree is updated in line with the subgoal package state is based on the goal ids. HOL Light's existing datatype for goals is extended to carry such an identity number. These identity-carrying goals are called *xgoals*.

```
type goalid = int;;
type xgoal = goal * goalid;;
```

Tactics are adjusted, or *promoted*, to work with xgoal inputs and outputs. These promoted tactics, called *xtactics*, have a datatype that is a trivial variant of HOL Light's original, with xgoals instead of goals. Proofs are performed using xtactics in place of tactics. The pretty printer for xgoals in written to ignore the goal id and print xgoals like goals, so that users see normal feedback when performing subgoal package proofs.

```
type xgoalstate = xgoal list * justification;;
type xtactic = xgoal →xgoalstate;;
```

The implementation of a promoted tactic first involves breaking up its xgoal argument into the original unpromoted goal and its goal id. The original, unpromoted tactic then gets applied to the unpromoted goal to result in a list of new subgoals and a justification function. These new subgoals are promoted into xgoals with unique ids, which then get inserted as branches of the node in the proof tree at the location determined by the input goal's id, along with abstract ML text for the tactic, based on the tactic's name. The promoted new subgoals and the justification function are returned as the result of the xtactic.

Rather than individually promote each tactic, we write a generic wrapper function for automatically promoting a supplied tactic of a given ML type, that takes the name of the tactic and the tactic itself as arguments. Below is the wrapper function for basic tactics that take no other arguments. The local value obj captures the abstract ML text of the tactic (see Sect. 5.3).

```
let tactic_wrap name (tac:tactic) : xtactic =
  fun (xg:xgoal) →
  let (g,id) = dest_xgoal xg in
  let (gs,just) = tac g in
  let obj = Mname name in
  let xgs = extend_gtree id (Gatom obj) gs in
  (xgs,just );;
```

This wrapper gets used to overwrite unpromoted tactics with their promoted versions, so that existing proof scripts can be replayed without adjustment.

```
let REFL_TAC = tactic_wrap"REFL_TAC" REFL_TAC;;
let STRIP_TAC = tactic_wrap "STRIP_TAC" STRIP_TAC;;
```

It is necessary to write wrapper functions for each ML type of tactic that can occur in a proof script. As the datatypes become more complex, so does the implementation of their corresponding wrapper functions. Slightly more complex than tactic_wrap is term_tactic_wrap, a function for promoting tactics that take a term argument. Its implementation is similar to tactic_wrap, except that here obj has the expression syntax of an ML name binding applied to a HOL term, enabling a refactored proof to refer to the term argument supplied in the proof.

```
let term_tactic_wrap name (tac:term→tactic) : term→xtactic =
  fun (tm:term) (xg:xgoal) →
  let (g,id) = dest_xgoal xg in
  let (gs,just) = tac g in
  let obj = Mapp (Mname name, [Mterm tm]) in
  let xgs = extend_gtree id (Gatom obj) gs in
  (xgs,just );;
```

Similar wrapper functions can be written for tactics that take other basic arguments, such as integers, strings or HOL types, or even structures of such arguments, such as a list of terms. Each time, the form of the obj local value varies to reflect the ML text of the tactic written with its arguments.

5.3 The Abstract ML Datatype

We use a datatype for capturing the ML text of how a tactic is written in a proof script, so that we can print out our refactored proof. We want this datatype to be capable of representing all the ML expression forms that commonly occur in tactic proofs, and to represent these as abstract syntax, so that we can easily perform syntax-based operations on these expressions when needed. Thus we define our recursive mlobject datatype. It is capable of referring to ML binding names, literals of basic datatypes (such as integers, strings, booleans, HOL terms and HOL types), structures (such as lists and tuples), function application, anonymous functions and local variables. It also supports the function composition operator separately, since this is often treated specially.

```
type mlobject =
    Mname of string
  | Mint of int
  | Mstring of string
  | Mbool of bool
  | Mterm of term
  | Mtype of hol_type
  | Mtuple of mlobject list
  | Mlist of mlobject list
  | Mapp of mlobject * mlobject list
  | Mfcomp of mlobject list
  | Mlambda of lvarid * mltype * mlobject
  | Mlvar of lvarid * mltype;;
```

5.4 Capturing Theorems

HOL theorems have no literal representation in the mlobject datatype. This is because, unlike for HOL types and HOL terms, printing the value of a theorem in an outputted proof script is of no use to the user, because it cannot be parsed in to recreate the theorem (in LCF-style theorem provers, at least). Instead it is necessary to print a theorem using its ML binding name, if it has one, or otherwise the ML text of its proof. Thus the theorem datatype thm is extended to carry abstract ML text. We call these extended theorems xthms. Like for xgoals, the pretty printer for xthms ignores the extension and prints an xthm in the same way as a thm, so that users see normal theorems when they view xthms.

```
type xthm = thm * mlobject;;
```

Each existing named theorem is promoted to have the xthm datatype, with its ML binding name for its abstract ML text, using the following wrapper:

```
let thm_wrap name (th:thm) : xthm =
  let obj = Mname name in
  mk_xthm (th,obj);;
```

All functions that take or return theorems also need to be promoted to work with xthms. We write wrapper functions for promoting any such functions of a given type, in a similar way to the wrapper functions for promoting tactics, although xthm wrappers are simpler to write because they do not need to incorporate their results into the proof tree.

```
let term_rule_wrap name (r:term→thm→thm) : (tm→xthm→xthm) =
  fun (tm:term) (xth:xthm) →
  let (th0,obj0) = dest_xthm xth in
  let th = r tm th0 in
  let obj = Mapp (Mname name, [Mterm tm; obj0]) in
  mk_xthm (th,obj);;
```

5.5 Automated Promotion

To avoid the user having to adapt proof scripts to explicitly promote ML values, Tactician automatically promotes all values. For values existing in the ML session when Tactician is loaded, this is done by executing a promotion function for each value in the OCaml environment, which is available to the user via the OCaml `Toploop` module and use of `Obj.magic`.[1] For values entered after Tactician has loaded, this is done by adjusting the OCaml toploop to execute a hook each time a value is added in the session.

6 Experiences and Limitations

Tactician has been tested on tens of thousands of lines of proof script files from the HOL Light theory libraries, including the Multivariate library, and the Flyspeck project. Tests involve running a proof script through a HOL Light session

[1] This is similar to a trick used in HOL Light to capture the theorem values in the ML session, implemented in HOL Light's `update_database.ml`.

with Tactician loaded, exporting the refactored proof script from the session, and replaying the exported script in a separate HOL Light session.

In typical usage, Tactician comfortably handles the large (30 lines) and very large (100+ lines) proof scripts found in HOL Light and Flyspeck. For example `NADD_COMPLETE` (100 lines) from the standard theory library's `realarith.ml`, and `PROPER_MAP` (115 lines) from Multivariate's `topology.ml`. However, processing thousand-line proof script files often fails, not due to the size of the script, but due to the likelihood of hitting one of the four current limitations of Tactician. We list the limitations in decreasing order of their frequency of occurrence:

Concrete Manipulation of the Goal. Since Tactician works on a different concrete goal datatype, it cannot process ML that uses the usual concrete constructors and destructors that manipulate goals as ML pairs (e.g. `fst` and `snd`). These need to be replaced with the abstract goal constructors and destructors provided by Tactician (e.g. `goal_hyp` and `goal_concl`).

ML Type Annotations. Tactician cannot process ML with type annotations for the "promotable" datatypes (i.e. `thm`, `conv`, `goal`, `tactic`, etc.). Any such type annotations must either have such datatypes changed to their promoted equivalents (i.e. `xthm`, `xconv`, `xgoal`, `xtactic`, etc.), or be removed.

Promotion of Obscure ML Datatypes. Tactician cannot automatically promote bindings with some obscure ML datatypes. However, these are very rarely encountered, and it is easy to write new promotion functions for them.

Promotion of "Unpromotable" ML Datatypes. Tactician cannot properly promote bindings that take a function as an argument if the function does not return a promotable datatype. The only common example of this in HOL Light is `PART_MATCH`, whose first argument is a term-to-term function. It outputs "`<???>`" in place of this argument, and warns the user.

We have found that it is always easy to overcome these limitations, either by manually adjusting the files, or, in the case of obscure ML datatypes, to extend Tactician. This usually takes about a minute per thousand-line file that fails, or a few minutes for multi-thousand line files. Tactician has been enhanced over the past few years to solve other limitations, and there is prospect of further enhancement to reduce further the frequency of hitting limitations. For example, the ML type annotations limitation can be solved by giving promoted datatypes the same name as their unpromoted equivalents, since there is no need for the unpromoted datatypes at the user level once Tactician has been loaded.

7 Conclusions

Tactic proofs are still used extensively in theorem proving, including in recent ground-breaking projects. However, there is little to help users refactor tactic proof scripts for maintenance or understanding. Tactician is a tool designed specifically to cater for this need.

In this paper we have explained the basics of how Tactician works, including how it captures proofs in memory, and how it transforms these proofs into suitable refactored forms. These facilities have been shown to work on the largest proof scripts from the HOL Light theory libraries and the Flyspeck project, although four current limitations mean that it is often necessary to perform a minute or two of manual preparation on large proof script files.

The principles explained have been implemented for the HOL Light system, but also apply to any implementation of the subgoal package. Most parts of Tactician should be straightforward to port to other systems, although the automatic promotion mechanism is specific to OCaml implementations, and adaptions to the subgoal package main functions are specific to HOL Light.

References

1. Adams, M., Aspinall, D.: Recording and refactoring HOL light tactic proofs. In: Workshop on Automated Theory eXploration, in Association with the 6th International Conference on Automated Reasoning (2012)
2. Arthan, R., Jones, R.: Z in HOL in ProofPower. In Issue 2005-1 of the British Computer Society Specialist Group Newsletter on Formal Aspects of Computing Science (2005)
3. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Math. Comput. Sci. **3**(3), 309–330 (2010). Springer
4. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science an EATCS Series. Springer, Heidelberg (2004)
5. Hales, T., Adams, M., Bauer, G., Dat, T.D., Harrison, J., Truong, H.L., Kaliszyk, C., Magron, V., McLaughlin, S., Thang, N.T., Truong, N.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., An, T.H., Trung, T.N., Diep, T.T., Urban, J., Ky, V.K., Zumkeller, R.: A Formal Proof of the Kepler Conjecture. arXiv:1501.02155v1 [math.MG]. arxiv.org (2015)
6. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009)
7. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., EngelHardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS Kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220. ACM (2009)
8. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
9. Paulson, L.: Logic and Computation: Interactive proof with Cambridge LCF. Cambridge University Press, Cambridge (1987)
10. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
11. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)
12. Tactician homepage: http://www.proof-technologies.com/tactician/