Towards Formal Proof Script Refactoring

Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov

CISA, School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland

Abstract. We propose proof script refactorings as a robust tool for constructing, restructuring, and maintaining formal proof developments. We argue that a formal approach is vital, and illustrate by defining and proving correct a number of valuable refactorings in a simplified proof script and declarative proof language of our own design.

1 Introduction

Theorem proving in-the-small is popular for investigating small domain-specific logics and for teaching logic. With maturing technology, theorem proving in-the-large is becoming more common, with big formalisations built in both industry and academia. Notable examples in verification include a specification of the IP stack [15] and functional correctness of a microkernel [9]. Formalised mathematics is also feasible: Gonthier formalised the proof of the Four Colour Theorem (FCT) [7] and Hales has an ongoing project to formally prove the Kepler Conjecture [8]. The completed proofs range in length from around 10,000 lines to 200,000 lines, each proving hundreds or thousands of lemmas and representing several person-years of work.

It is encouraging that large developments are possible, but they are far from easy. Writing proofs is often harder than writing programs. Formal proofs are more complex, dense, and interdependent than similarly sized programs. Yet they are developed with primitive tools akin to those used for programming in the 1960s: often little more than basic text editors, with no high-level means of rapid construction, easy modification or browsing. Lack of modern support for proof construction and maintenance is a main reason that interactive provers are not used more widely; it makes them incredibly tedious to learn and use.

By contrast, maintenance of large programs is well supported by modern Software Engineering (SE) tools. Software systems of hundreds of thousands of lines of code can be managed with relative ease. One important SE technique is *refactoring*. A *refactoring* is a semantics-preserving transformation of code which improves design, structure or readability [6, 12]. It may be pervasive, but routine. Ideally refactorings are tool-assisted: an algorithm checks safety pre-conditions before making global changes in one go. Many refactorings are simple operations with complex pre-conditions, e.g., *Rename* and *Move*. Our hypothesis is that by adapting and extending refactoring techniques from SE we can make development and maintenance of formal proof scripts easier and more accessible

to new users, as well as more productive for expert users. As anecdotal evidence to motivate our work, we note that Gonthier mentions, in [7], having to spend a number of months refactoring his FCT development by hand.

Ensuring correctness of proof refactorings is vital because proof scripts can take arbitrarily long to re-check, and unexpected changes to lemma statements can change the *meaning* of a development. It is also non-trivial; for example, complex tactics make analysis of dependencies difficult, as noted in [13], and notions of semantics for fully-blown proof scripts have not been well studied compared with programming language semantics. Even refactoring tools for programming languages, which have been studied for almost 20 years, are full of bugs [5].

In this work we study refactoring formally in a simple, generic proof script language of our own design in order to understand and overcome the main challenges. In particular, we use Hiproofs [4] as a generic notion of proof as it provides a clean theoretical base on which to build; furthermore, we believe the hierarchy offers opportunities for refactoring and proof understanding. We do not intend this work to cover all aspects of a practical implementation, but use it as an exploratory study into the viability, applicability, and challenges associated with refactoring.

Contributions. Our two main contributions are a generic proof script language, with a declarative proof language, and a formal treatment of a number of proof refactorings. Firstly, we give a formal semantics to the proof scripts and prove that declarative proofs construct valid proofs. In particular, we formalise a notion of *gaps* in a proof. Secondly, we define what we mean by proof script refactoring, and the appropriate notion of semantics preservation, and define several valuable refactorings, including *rename lemma* and *backward to forward*, which transforms a backwards-style proof into a forward-style one. Finally we prove that these refactorings are correct in some meaningful sense.

Outline of paper. In the next section, we introduce Hiproofs, as a representation for proof, and Hitac as an idealised tactic language on which we base our work. In Section 3 we introduce the proof script language, its semantics, and give a formal definition of proof script refactoring. We then, in Section 4, describe the declarative proof language and give an example of our proof scripts in Section 5. Section 6 describes a number of refactorings and we conclude in Section 7.

2 Background

Hiproofs are a hierarchical representation of the proof trees constructed by tactics. The hierarchy makes explicit the relationship between tactic calls and the proof tree constructed by these tactics. Hiproofs were first investigated by Denney et al in [4] and can be given a denotational semantics as a pair of forests, viewed as posets. One partial order provides a notion of hierarchy, the other of sequential composition. An abstract example of a Hiproof is given in Figure 1, where a, b, and c are called *atomic tactics* i.e. black boxes.

Figure 1 reads as follows: at the top, the abstract tactic l first applies an atomic tactic a. The tactic a produces two subgoals, the first of which is solved

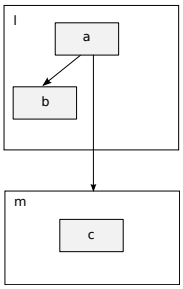

Fig. 1. A Hiproof

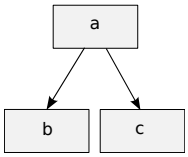

Fig. 2. The skeleton

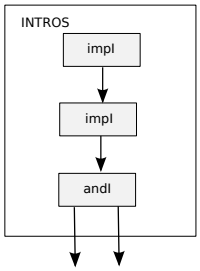

Fig. 3. INTROS

by the atomic tactic b within the application of l. Thus, the high-level view is tactic l produces a single subgoal, which is then solved by the tactic m. The underlying proof tree, called the *skeleton*, is shown in Figure 2. Conditions placed upon construction of Hiproofs ensure that they can always be *unfolded* into the skeleton. More concretely, Figure 3 shows the application of an *INTROS* tactic as a Hiproof; the trailing edges are goals that must be solved by composing other Hiproofs. In [1], Aspinall et al gave an operational account of Hiproofs, based on *derivation systems*. They also introduced a tactic language, Hitac, which constructs Hiproofs. We describe this work now, as it provides the basis for what follows.

Derivation Systems. In this work we will not commit to a specific logical system; instead, we work within a *derivation system*, which can be thought of as a simple logical framework. It defines sets of *atomic goals* $\gamma \in \mathcal{G}$ and *atomic tactics* $a \in \mathcal{A}$. What we call an *atomic goal* is a judgement form in the underlying derivation system, and what we call an *atomic tactic* is an inference rule schema:

$$\frac{\gamma_1 \cdots \gamma_n}{\gamma}\ a \in \mathcal{A}$$

stating that the atomic tactic a, given subgoals $\gamma_1, \ldots, \gamma_n$, produces a proof of γ. We do not formalise how the rule schemata and instances are related.

Hiproofs. The concrete syntax of Hiproofs is defined by the grammar:

$$s \ ::= \ a \mid id \mid [l]s \mid s\ ;\ s \mid s \otimes s \mid \langle\rangle$$

Sequencing ($s\ ;\ s$) corresponds to composing boxes by arrows, tensor ($s \otimes s$) places boxes side-by-side, and labelling ($[l]s$) introduces a new labelled box. Identity (id) and empty ($\langle\rangle$) are units for ; and $\otimes$ respectively. Labelling binds weakest, then sequencing with tensor binding most tightly. We can now give a syntactic description of the Hiproof in Figure 1: $([l]\,a\ ;\ b \otimes id)\ ;\ [m]\ c$.

We say that a Hiproof is *valid* if it is well-formed and if atomic tactics are applied correctly. This notion is formalised with a validation relation $s \vdash g_1 \longrightarrow g_2$, where g_i are lists of goals. This relation, defined below, states that the Hiproof s

is a proof of g_1 with remaining subgoals g_2. Thus, Hiproofs can represent partial proofs. The γ_i are single goals and @ is list append.

$$\frac{\frac{\gamma_1 \dots \gamma_n}{\gamma}\ a \in \mathcal{A}}{a \vdash [\gamma] \longrightarrow [\gamma_1, \dots, \gamma_n]}\ \text{(V-Atomic)} \qquad \frac{s_1 \vdash g_1 \longrightarrow g \qquad s_2 \vdash g \longrightarrow g_2}{s_1\ ;\ s_2 \vdash g_1 \longrightarrow g_2}\ \text{(V-Seq)}$$

$$\frac{s \vdash [\gamma] \longrightarrow g}{[l]s \vdash [\gamma] \longrightarrow g}\ \text{(V-Label)} \qquad \frac{s_1 \vdash g_1 \longrightarrow g_1' \qquad s_2 \vdash g_2 \longrightarrow g_2'}{s_1 \otimes s_2 \vdash g_1 @ g_2 \longrightarrow g_1' @ g_2'}\ \text{(V-Tens)}$$

$$\frac{}{\langle\rangle \vdash [] \longrightarrow []}\ \text{(V-Empty)} \qquad \frac{}{id \vdash [\gamma] \longrightarrow [\gamma]}\ \text{(V-Id)}$$

Hitac. We extend the Hitac grammar from [1] with *lemma application*:

$$\begin{array}{rcl} t & ::= & a \mid id \mid [l]t \mid t\ ;\ t \mid t \otimes t \mid \langle\rangle \\ & \mid & assert\ \gamma \\ & \mid & t \mid t \\ & \mid & name(t, \dots, t) \\ & \mid & X \\ & \mid & lem\ l \end{array}$$

In addition to the standard Hiproof constructs, goal assertions ($assert\ \gamma$) can control the flow; alternation ($t \mid t$) allows choice; and, defined tactics ($name(t, \dots, t)$) and variables (X) allow us to build recursive tactic programs. Tactic evaluation is defined relative to a *proof environment*: $(\mathcal{T}, \mathcal{L})$. The lemma environment, $\mathcal{L} : name \rightarrow (goal \times s)$, is a map from lemma names to a goal and Hiproof pair; the tactic environment, $\mathcal{T} : name \rightarrow (\overline{X}, t)$, maps tactic names to their parameter list and Hitac tactic.

Evaluation of a tactic is defined as a relation $\langle g, t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle$, which should be read as: 'the tactic t applied to the list of goals g returns a Hiproof s and remaining subgoals g', under $(\mathcal{T}, \mathcal{L})$'. We give the evaluation rules for *tensor*, *lemma application*, and *defined tactics* below, where we write $\overline{X}$ as shorthand for the list $[X_1, \dots, X_n]$. For a full presentation of the semantics and a proof of Theorem 1, please refer to [1].

$$\frac{\langle g_1, t_1\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1, g_1'\rangle \qquad \langle g_2, t_2\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_2, g_2'\rangle}{\langle g_1 @ g_2, t_1 \otimes t_2\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1 \otimes s_2, g_1' @ g_2'\rangle}\ \text{(B-Tac-Tens)}$$

$$\frac{\mathcal{T}(name) = (\overline{X}, t) \qquad \langle g, t[t_1/X_1, \dots, t_n/X_n]\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle}{\langle g, name(t_1, \dots, t_n)\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle}\ \text{(B-Tac-Def)}$$

$$\frac{\mathcal{L}(l) = (\gamma, s) \qquad s \vdash \gamma \longrightarrow []}{\langle \gamma, lem\ l\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle [lem\ l]s, []\rangle}\ \text{(B-Tac-Lem)}$$

Theorem 1 (Correctness of big-step semantics) *If* $\langle g_1, t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g_2\rangle$ *then* $s \vdash g_1 \longrightarrow g_2$.

3 Proof Scripts

Proof scripts allow us to build and extend proof environments. For simplicity, proof scripts currently only consist of tactic definitions and lemmas. The grammar for proof scripts is:

$$\begin{array}{lcl} \mathit{proofscript} & ::= & \textbf{script}\ \mathit{name} \\ & & \quad \mathit{scriptobj}^* \\ & & \textbf{end} \end{array} \qquad \begin{array}{lcl} \mathit{scriptobj} & ::= & \textbf{begin} \\ & | & \textbf{tacdef}\ \mathit{name}(X_1, \ldots, X_n) := t \\ & | & \textbf{lemma}\ \mathit{name}\textbf{:}\ \mathit{goal} \\ & & \quad \mathit{prf} \end{array}$$

Proof scripts essentially consist of a sequence of lemmas and tactics, within **begin** and **end** tags. Lemmas are named and consist of a goal and a formal proof *prf*; parameterised tactics can be defined, where the X_i are the tactic variables occurring within t and must be instantiated when the tactic is used.

Top level evaluation of proof scripts is defined by:

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \textbf{script}\ \mathit{name}\ \mathit{scriptobjs}\ \textbf{end} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}\ \text{(PS-Script)}$$

where the judgement $\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$, which operates on a sequence of script objects, is defined in Figure 4. The evaluation relation states that the script *proofscript* can be evaluated resulting in an *environment* $\langle \mathcal{T}, \mathcal{L} \rangle$. When we do not need to explicitly refer to the environment, we will write $\vdash \mathit{proofscript}$ to mean $\exists\ \mathcal{T}\ \mathcal{L}.\ \vdash \mathit{proofscript} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$. We say a proof script is *well-formed* iff $\vdash \mathit{proofscript}$. Well-formedness does not imply that the proofs are complete: there may be gaps; *proof checking* is a low-level process that would ensure the Hiproofs constructed have no trailing edges. For brevity, we often use P to refer to proof scripts.

$$\vdash \textbf{begin} \Downarrow \langle \{\}, \{\} \rangle \qquad \text{(PS-Begin)}$$

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad n \notin \mathit{names}(\mathit{scriptobjs}) \qquad \langle \gamma, \mathit{prf} \rangle \Downarrow_{(\mathcal{T}, \mathcal{L})} \langle s \rangle}{\vdash \mathit{scriptobjs}\ \textbf{lemma}\ n\textbf{:}\ \gamma\ \mathit{prf} \Downarrow \langle \mathcal{T}, \mathcal{L}[n \mapsto (\gamma, [n]s)] \rangle}\ \text{(PS-Lem)}$$

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad n \notin \mathit{names}(\mathit{scriptobjs}) \qquad \mathit{variables}(t) \subseteq \overline{X} \qquad \mathit{scriptobjs} \vdash t}{\vdash \mathit{scriptobjs}\ \textbf{tacdef}\ n(\overline{X}) := t \Downarrow \langle \mathcal{T}[n \mapsto (\overline{X}, [n]t)], \mathcal{L} \rangle}\ \text{(PS-Tac)}$$

Fig. 4. Proof script evaluation

The rule PS-Begin ensures scripts start with a **begin** and initialises an empty environment. Tactics (or lemmas) extend $\mathcal{T}$ (or $\mathcal{L}$) as long as they satisfy the preconditions. The functions *names* and *variables* are defined recursively on

scripts and tactics and the relation $scriptobjs \vdash t$ checks for well-formedness of tactics: ensuring that only tactics and lemmas defined above it in the script can be used. The notation $\mathcal{T}[n \mapsto (\overline{X}, [n]t)]$ means extending the map, $\mathcal{T}$, by adding an element. For a lemma, the important precondition is that $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$. This is the *proof relation*, which states that when you apply *prf* to the goal γ it results in a Hiproof, s. We will instantiate *prf* in the next section, but it could be any proof language associated with a *valid* proof relation i.e. one that constructs valid Hiproofs. Both lemmas and tactics are labelled with their name when added to the environment allowing us to see, in the hierarchy, where tactics and lemmas have been applied.

3.1 Proof Script Refactoring.

We first define a more general notion of proof script transformation as follows:

Definition 1 (Proof Script Transformation) *A pair $(\mathcal{P}, \mathcal{O})$, where $\mathcal{P}$ is a predicate, called the precondition, and $\mathcal{O} : proofscript \to proofscript$, is a **proof script transformation** if, for all scripts P, such that $\vdash P$ and $\mathcal{P}(P)$, we have $\vdash \mathcal{O}(P)$.*

That is, applying a proof script transformation, as long as the precondition $\mathcal{P}$ holds, guarantees that it preserves well-formedness.

We take the view that the statements proved within lemmas are the key semantics objects in a script, motivating the following definition:

Definition 2 (Statement Preservation) *A proof script transformation, $(\mathcal{P}, \mathcal{O})$, is **statement preserving** if for all scripts P such that $\mathcal{P}(P)$ and $\vdash P \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$ then $\vdash \mathcal{O}(P) \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$ and we have:*

$$\forall\ l \ if\ \ \mathcal{L}(l) = (\gamma, s)\ \exists\ l'\ s'\ s.t.\ \mathcal{L}'(l') = (\gamma, s').$$

That is, we prove at least the same statements before and after a transformation. Thus, we have:

Definition 3 (Proof Script Refactoring) *A **proof script refactoring** is a proof script transformation that is statement preserving.*

4 A Declarative Proof Language

To explore the refactoring possibilities within proofs, we describe a declarative proof language and give it a semantics which constructs valid Hiproofs. We describe *prf* with the following grammar:

prf	::=	**proof(** *rule* **)**	*stmt*	::=	$\langle\rangle$
		stmt*		|	[*name*]**:{** *prf* **}**
		qed		|	**apply** *rule*
	|	**gap**		|	**show** *name* : *goal prf*
				|	**have** *name* : *goal prf*
rule	::=	*t*		|	**from** *name** **show** *goal* **by** *rule*

The core component of the language is a *proof block*: **proof(***rule***)** *stmts* **qed**. Proof blocks operate on a single goal, applying the initial rule before solving the resulting subgoals by the statements inside it. The key statement is **show**, which solves the goal it is applied to. The *empty* statement $\langle\rangle$ operates on an empty list, finishing off a proof; tactics can be applied directly with **apply**; forward proof is possible by using **have** to extend the environment then the **from**...**show**...**by** construct to perform the forward step. Hierarchy can be added, using the labelling construct: [*name*]**:{** *prf* **}**; finally, goals can be skipped with the **gap** command. Syntactic conveniences, for example, **by** *rule* $\equiv$ **proof(***rule***)** $\langle\rangle$ **qed** can be introduced.

In Figure 5, we give a big-step semantics to declarative proofs with the relation $\langle g_1, prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle$. A proof *prf* applied to a list of goals g_1 will result in a Hiproof s. Top level *prf* evaluations always operate on a single goal, and the evaluation rules in Figure 5 reflect this. Statement lists are evaluated one at a time; the statement being operated on directly is highlighted. We use :: to refer to the *cons* list constructor.

$$\frac{\langle[\gamma], t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1, g\rangle \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle[\gamma], \textbf{proof(}t\textbf{)}\ stmts\ \textbf{qed}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\ ;\ s_2\rangle} \text{(B-Prf-Block)}$$

$$\langle[\gamma], \textbf{gap}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle id\rangle \text{(B-Prf-Gap)}$$

$$\langle[], \langle\rangle\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle\langle\rangle\rangle \text{(B-Prf-Empty)}$$

$$\frac{\langle[\gamma], prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle\gamma :: g, [l]\textbf{:\{}\ prf\ \textbf{\}}\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle([l]s_1)\ \otimes\ s_2\rangle} \text{(B-Prf-Lab)}$$

$$\frac{\langle g_1, t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1, g_2\rangle \qquad \langle g_2, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle g_1, \textbf{apply}\ t\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\ ;\ s_2\rangle} \text{(B-Prf-App)}$$

$$\frac{\langle[\gamma], prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle\gamma :: g, \textbf{show}\ name\colon \gamma\ prf\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\ \otimes\ s_2\rangle} \text{(B-Prf-Show)}$$

$$\frac{\langle[\gamma], prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad name \notin names(\mathcal{T}) \cup names(\mathcal{L}) \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[name \longmapsto (\gamma, s_1)])} \langle s\rangle}{\langle g, \textbf{have}\ name\colon \gamma\ prf\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle} \text{(B-Prf-Have)}$$

$$\frac{\begin{array}{c}\mathcal{L}(n_1) = (\gamma_1, s_1) \quad \ldots \quad \mathcal{L}(n_n) = (\gamma_n, s_n) \\ \langle[\gamma], t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s'\rangle\end{array}}{\begin{array}{c}\langle\gamma :: g, \textbf{from}\ n_1\ \ldots\ n_n\ \textbf{show}\ name\colon \gamma\ \textbf{by}\ t\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \\ \langle(s\ ;(s_1 \otimes \ldots \otimes s_n)) \otimes s', []\rangle\end{array}} \text{(B-Prf-From)}$$

Fig. 5. Declarative proof language big-step semantics

Proof blocks operate on singleton goal lists and are evaluated by executing the initial tactic, then feeding the resulting subgoals into the enclosed statements. The **gap** proof construct also operates on singleton goals, placing an *identity* Hiproof to feed the goal out. In B-PRF-HAVE, we see that the intermediate step is added to the proof environment. We enforce the new name to be unique, but this is not necessary: if we drop this restriction we can have local overloading in the current proof block. This does, however force us to provide more complex preconditions and transformation rules for the refactorings.

The rule B-PRF-FROM is the most complicated. The first set of preconditions check that the names used exist in the lemma environment; the next ensures that the tactic justification returns exactly the stated goals that the names refer to; the third simply evaluates the remaining statements. Finally the Hiproof is constructed by tensoring together all of the Hiproofs for each individual subgoal and placing them after the Hiproof resulting from the tactic application. In order to ensure that *valid* Hiproofs are constructed, we prove:

Theorem 2 (Soundness of big-step semantics) *If* $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*. Moreover, if* prf *is* ***gap-free*** *then* $s \vdash \gamma \longrightarrow []$*.*

Proof: We proceed by induction on the height of the derivations. For empty, the rule B-PRF-EMPTY matches V-EMPTY directly with $g = []$. Gaps are validated with V-ID and $g = [\gamma]$. For B-PRF-BLOCK, Theorem 1 and the induction hypothesis allow us to apply V-SEQ. Similarly, with B-PRF-SHOW we use the induction hypothesis twice and V-TENS, with g being the concatenation of both. The other cases are similar. When the proof is gap-free, we note that B-PRF-GAP is the only base case to introduce a discrepancy between Hiproof validation and tactic evaluation, thus g must be []. In fact, g is exactly the 'gapped' goals.

Theorem 3 (Completeness of big-step semantics) *If* $s \vdash \gamma \longrightarrow []$ *for a given environment* $(\mathcal{T}, \mathcal{L})$ *then there exists a gap-free* prf *such that* $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$.

Proof: If s is a Hiproof such that $s \vdash \gamma \longrightarrow []$ then, trivially, '**by** s' works when we consider the Hiproof as a Hitac tactic.

5 Example

In order to make these languages more concrete, we provide a small example proof script. Space restrictions require any such example is necessarily trivial but we hope it conveys some of the main features. We instantiate a derivation system with first order logic, with atomic tactics given by the well-known natural deduction rules, a few of which are given below:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ impI \qquad \frac{\Gamma \vdash P \qquad \Gamma, Q \vdash R}{\Gamma, P \to Q \vdash R}\ impE \qquad \frac{}{\Gamma, P \vdash P}\ ax \qquad \frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R}\ conjE$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ conjI \qquad \frac{\Gamma \vdash P[x := x_0]}{\Gamma \vdash \forall\ x.P}\ allI \qquad \frac{\Gamma, P[x := t] \vdash Q}{\Gamma, \forall\ x.P \vdash Q}\ allE$$

An example script, defining two tactics and proving two lemmas is given in Figure 6. Note that we elide the empty statement in both lemmas.

```
script example begin
  tacdef REPEAT(X) := X ; (REPEAT(X) | id)
  tacdef intros := REPEAT(impI | allI | conjI)

  lemma lemma1 : ⊢ P ∧ Q → Q ∧ P
  proof(intros)
   show q : {P ∧ Q} ⊢ Q by (conjE ; ax)
   show p : {P ∧ Q} ⊢ P by (conjE ; ax)
  qed

  lemma lemma2 : ⊢ (∀ x. P x → Q x) → (∀ x. P x) → (∀ x. Q x)
  proof(intros)
   show {∀ x. P x → Q x, ∀ x. P x} ⊢ Q x
   proof(REPEAT(allE))
    show {P x → Q x, P x} ⊢ Q x by impE ; (ax ⊗ ax)
   qed
  qed
end
```

Fig. 6. Example proof script

6 Refactorings

We are now ready to refactor proof scripts. We describe *rename lemma*, *swap objects*, *transform proof*, *backwards proof to forwards proof*, and *extract subproof* in some detail. We summarise, in Figure 7, the main refactorings we have considered thus far. Finally in Section 6.6 we show how Figure 6 could be refactored.

6.1 Rename Lemma

Renaming a lemma may seem like a trivial action but, if that lemma has been applied multiple times in a proof development, the new name must be propagated and must not clash with any other names in the development. This makes it a tedious and error-prone task for humans. The refactoring takes three parameters: a script to operate on, an old lemma name, o, and a new lemma name, n.

Preconditions. In common with all other refactorings that we are currently considering, *rename lemma* has the precondition that the proof script P that it acts on must be well-formed i.e. $\vdash P$. We must also ensure that we have no name clashes with the new name: $n \notin names(P)$.

Transformation rules. We define this transformation on the structure of proof scripts. There are three classes of rules operating on a *proofscript*, a *prf*, or a

Generalise Tactic	Replace a *subtactic* with a var, creating a more general tactic.
Fold/Unfold Proof	Declarative proofs can be collapsed into raw tactic applications and vice-versa.
Fold/Unfold Tactic	Fold or unfold a defined tactic.
Fill Gap	Replace a **gap** with a proof that solves the goal.
Add/Rem Hierarchy	Introduce or remove labelled boxes to a proof.
Safe Delete	Delete a lemma or tactic as long as it is not used.
Copy	Copy a lemma or tactic.
Rename	Rename a lemma or tactic.
Backward to Forward	Convert a backward proof into a forward proof.
Have to Lemma	Lift a **have** statement up to the status of lemma.
Extract Subproof	Extract a subproof of a goal into a lemma.

Fig. 7. Summary of refactorings

t. We give examples of the action on *proofscript* and t in Figures 8 and 9. The *rename tactic* refactoring is analagous, except we must take into account possibly recursive tactics.

$$\frac{\mathit{scriptobjs} \xrightarrow{rl(o,n)} \mathit{scriptobjs}'}{\textbf{script}\ \mathit{name}\ \mathit{scriptobjs}\ \textbf{end} \xrightarrow{rl(o,n)} \textbf{script}\ \mathit{name}\ \mathit{scriptobjs}'\ \textbf{end}}$$

$$\textbf{begin} \xrightarrow{rl(o,n)} \textbf{begin}$$

$$\mathit{scriptobjs}\ \textbf{lemma}\ o\textbf{:}\ \gamma\ \mathit{prf} \xrightarrow{rl(o,n)} \mathit{scriptobjs}\ \textbf{lemma}\ n\textbf{:}\ \gamma\ \mathit{prf}$$

$$\frac{\mathit{scriptobjs} \xrightarrow{rl(o,n)} \mathit{scriptobjs}' \qquad \mathit{prf} \xrightarrow{rl(o,n)} \mathit{prf}' \qquad o \neq ln}{\mathit{scriptobjs}\ \textbf{lemma}\ ln\textbf{:}\ \gamma\ \mathit{prf} \xrightarrow{rl(o,n)} \mathit{scriptobjs}'\ \textbf{lemma}\ ln\textbf{:}\ \gamma\ \mathit{prf}'}$$

$$\frac{\mathit{scriptobjs} \xrightarrow{rl(o,n)} \mathit{scriptobjs}' \qquad t \xrightarrow{rl(o,n)} t'}{\mathit{scriptobjs}\ \textbf{tacdef}\ tn(\overline{X}) := t \xrightarrow{rl(o,n)} \mathit{scriptobjs}\ \textbf{tacdef}\ tn(\overline{X}) := t'}$$

Fig. 8. Script level transformations

Correctness. We want to prove that this operation is indeed a refactoring:

Theorem 4 (Rename Lemma Correctness) *If, for a proof script P and old and new names o and n that satisfy the preconditions above and*

$$P \xrightarrow{rl(o,n)} P' \qquad \textit{and} \qquad \vdash P \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{then} \quad \vdash P' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

and we have: $\forall\ l$ if $\mathcal{L}(l) = (\gamma, s)\ \exists\ l'\ s'$ s.t. $\mathcal{L}'(l') = (\gamma, s')$.

$$\frac{t_1 \xrightarrow{rl(o,n)} t_1' \qquad t_2 \xrightarrow{rl(o,n)} t_2'}{t_1 \ ; \ t_2 \xrightarrow{rl(o,n)} t_1' \ ; \ t_2'}$$

$$\frac{l \neq o}{lem\ l \xrightarrow{rl(o,n)} lem\ l}$$

$$lem\ o \xrightarrow{rl(o,n)} lem\ n$$

Fig. 9. Tactic level transformations

Proof. We prove that $\vdash P' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$ by induction on the transformation rules. For each rule, we show that if the script evaluates before the rule is applied, then it evaluates to an equivalent environment afterwards; this motivates our need for the precondition as it is required as a premiss for one of the evaluation rules. We can then see that $\mathcal{L}'$ satisfies the statement preservation property.

6.2 Swap Objects

We can swap two adjacent objects if they have no dependency. In Figure 6 we can swap the two lemmas, but not the definitions of *intros* and *REPEAT*. We can repeat this refactoring to get the more general *move object* refactoring. To simplify presentation, we assume that we are swapping two adjacent lemmas (although the general refactoring covers all four cases). *Swap object* takes two parameters: the name of a lemma, x and the script P. We take the convention that the named lemma is to be moved up one place.

Preconditions. Given $posn(x,\ P) = n$, $objAt(n-1,\ P) = y$, and $envAt(n,\ P) = (\mathcal{T}, \mathcal{L})$. If $proof(x) = prf$, $\mathcal{L}(x) = (\gamma, s)$, and $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ then we must have $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T}, del(y,\ \mathcal{L}))} \langle s \rangle$.

That is, the lemma x can still be proved in an environment without y. This formulation of the preconditions simplifies the correctness proof, but it could also be described purely syntactically for our language; however, for languages with sophisticated tactics like *auto*, we would need to use the semantic information. All the functions used here are easily defined on scripts.

Transformation. We only show a couple of transformation rules:

$$\frac{obj_2 \ \ objs \xrightarrow{swap(x)} objs' \qquad nameOf(obj_2) \neq x}{obj_1 \ \ obj_2 \ \ objs \xrightarrow{swap(x)} obj_1 \ \ objs'}$$

$$obj_1 \ \ \textbf{lemma}\ \text{x:}\ \gamma\ prf \ \ objs \xrightarrow{swap(x)} \textbf{lemma}\ \text{x:}\ \gamma\ prf \ \ obj_1 \ \ objs$$

Correctness. We elide the proof that this is indeed a refactoring.

6.3 Transform Proof

This refactoring is an *enabling refactoring* for the ones to follow. Essentially, it takes a *proof transformation*, $\mathcal{R} : prf \rightarrow prf$, and a lemma name, n, as parameters and applies $\mathcal{R}$ to the proof of that lemma, leaving everything else untouched. The precondition for this refactoring is that the proof transformation preserves provability. We do not give the transformation rules here as they are straightforward. This is clearly a refactoring: the provability precondition matches directly with the premiss for P-Lem.

6.4 Backward Proof to Forward Proof

This refactoring transforms a proof *prf* that is in the form of Figure 10 to the form of Figure 11.

proof(t **)**
  show *goal1* : γ_1 prf_1
  ⋮
  show *goaln* : γ_n prf_n
qed

Fig. 10. Before

proof
  have *goal1* : γ_1 prf_1
  ⋮
  have *goaln* : γ_n prf_n
  from *goal1* ... *goaln* **show** γ **by** t
qed

Fig. 11. After

Preconditions. We have one precondition: there are no '**apply** t' steps within the proof block. Rather than a technical limitation, it simplifies the presentation of the rules below. In order to remove it, we would have to convert any **apply** steps into a declarative proof format, which is another refactoring.

Transformation rules. We describe the refactoring using a set of transformation rules, a subset of which is given below:

$$\frac{stmts \xrightarrow{b2f} stmts' \qquad [n_1, \ldots, n_n] = shows(stmts)}{\textbf{proof(}t\textbf{)}\ stmts\ \textbf{qed} \xrightarrow{b2f} \textbf{proof}\ stmts'\ \textbf{from}\ n_1\ \ldots\ n_n\ \textbf{show}\ \gamma\ \textbf{by}\ t\ \textbf{qed}}$$

$$\frac{stmts \xrightarrow{b2f} stmts'}{\textbf{show}\ name\textbf{:}\ \gamma\ prf\ \ stmts \xrightarrow{b2f} \textbf{have}\ name\textbf{:}\ \gamma\ prf\ \ stmts'}$$

This time, the transformation rules work only on a *prf*. In order to apply the refactoring at the script level, we use *transform proof*. The function *shows* constructs a list of all the goals.

Correctness. Since this refactoring applies only to the proof and the precondition for *transform proof* is that *backward to forward* preserves provability, we only need to show:

Theorem 5 (Provability Preservation of Backward to Forward) *If prf is a declarative proof of γ satisfying the preconditions of* backward to forward *then*

$$prf \xrightarrow{b2f} prf' \quad and \quad \langle \gamma, prf' \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle$$

Proof. Since **show** statements get transformed to **have** statements by the refactoring, and by the precondition that the names are fresh in the environment, we can guarantee that all the names are in the environment when the **from**...**show**...**by** is executed. This succeeds because the names map directly to the subgoals produced by t initially.

6.5 Extract Subproof

In this more complex refactoring, we show how a proof of a subgoal within a lemma can be extracted as a lemma in its own right. It is, in fact, a composition of two simpler refactorings:

Show to Have: transforms a **show** statement into a **have** statement and replaces the proof of the show statement with a '**by** *lem n*', where n is the user supplied name for the have statement.

Have to Lemma: moves a have statement up to the top level of the script: there are no preconditions and no change is required to the rest of the proof. This refactoring would be useful if an intermediate lemma is more widely applicable.

Figures 12 to 14 show how this refactoring proceeds. We do not give a more formal description here. It is worth noting that we can compose these refactorings because *Have to Lemma* does not have any preconditions (other than well-formedness); however, in general, to ensure that two refactorings can be composed to form another correct refactoring we must be able to prove that the preconditions for the second refactoring are always satisfied. In future work, we intend to investigate composition by appending *postconditions* to our notion of a refactoring.

6.6 Example Refactoring

Finally, using *rename lemma*, *fold tactic*, *backward proof to forward proof*, and *swap lemma* we can transform the proof script in Figure 6 into Figure 15. In particular, we rename *lemma1* to *conj_comm* and *lemma2* to *all_mp*, which better reflect their meaning, and swap their position. We have used *backward proof to forward proof* to transform *conj_comm* and also *fold tactic* to replace the identical applications of *conjE* ; *ax* with a named tactic called *conjEax*.

7 Related Work and Conclusions

This paper introduces proof script refactoring as a way to make structured changes to a proof development. We describe a number of valuable refactorings including *rename lemma*, and *extract subproof* for a simple proof script and

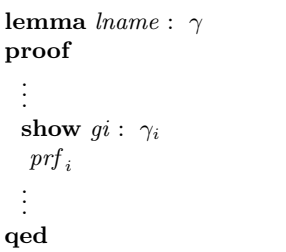

Fig. 12. Before

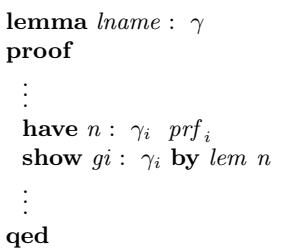

Fig. 13. Step one

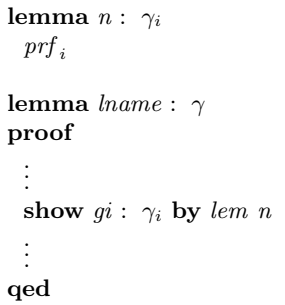

Fig. 14. After

declarative proof language. We believe that the formal approach we take is necessary: the time required for proof-checking and risking of changing the meaning of a lemma makes the correctness of refactorings vital.

While we believe our work on proof script refactoring is unique, there is a large literature in the domain of programming language refactoring. Fowler takes a test-based approach to refactoring in [6]; this book, widely considered to be the 'handbook of refactoring', consists of over 70 refactorings with a detailed description of the motivation for each refactoring and how to carry it out safely. We note that refactoring has benefited from formal study in prototypical languages: Cornélio et al specify refactorings for a subset of Java, called ROOL [3] and prove semantics preservation using a set of basic algebraic laws, expressing equivalences between objects. In [10], Li and Thompson discuss a formal specification of Haskell refactorings based on an abstract representation of a program and provide a proof that the semantics of the program are preserved during the refactoring. Mens and Tourwe give a much more detailed survey of programming language refactoring in [11]. There is also an interest in refactoring formal specifications. For example, in [16], the authors suggest refactorings for Z specifications based on experience on several large-scale projects. The effects of refactorings in Z are closely related to those in a formal proof script as, when schemas are refactored, this has an effect on all proofs relying of properties of these definitions. Schairer and Hutter describe a transformation framework for formal specifications in [14]. Although they do not consider semantics preservation, their approach is similar to our own, working independent of any logical system.

Declarative proof languages were pioneered by the Mizar system [17]. Our declarative language is designed to incorporate many of the core features of popular derivative languages, such as Isar for Isabelle [18], and C-zar for Coq [2]. However, due to our abstract approach, we do not have declarative statements that refer to the logical structure of a goal. For example, we only have **show** instead of the Isar-style **fix**...**assume**...**show** or the direct mapping between inference rules and C-zar statements such as **assume**. Our semantic approach

```
script example begin
  tacdef REPEAT(X) := X ; (REPEAT(X) | id)
  tacdef intros := REPEAT(impI | allI | conjI)
  tacdef conjEax := conjE ; ax

  lemma all_mp : ⊢ (∀ x. P x → Q x) → (∀ x. P x) → (∀ x. Q x)
  proof(intros)
   show {∀ x. P x → Q x, ∀ x. P x} ⊢ Q x
   proof(REPEAT(allE))
    show {P x → Q x, P x} ⊢ Q x  by impE ; (ax ⊗ ax)
   qed
  qed

  lemma conj_comm : ⊢ P ∧ Q → Q ∧ P
  proof
   have q : {P ∧ Q} ⊢ Q  by conjEax
   have p : {P ∧ Q} ⊢ P  by conjEax
   from q p show {P ∧ Q} ⊢ Q ∧ P by intros
  qed
end
```

Fig. 15. Refactored example proof script

to gaps is more closely related to Isar, where lemmas can be proved with gaps; C-zar, by comparison, does not allow a final 'qed' with unproven subgoals.

Further work. There are a number of extensions to our language we wish to consider. We would like to investigate a simple module system and define refactorings that operate at the module level. We could, for example, merge modules or move lemmas from one module to another. A more sophisticated *logical framework* would enable us to refactor statements directly, allowing us, for example, to *remove assumption*, if it is unused. Our proof script language also needs to be extended: we have yet to deal with *definitions* and *axioms*, both of which come with refactorings. On the practical side, we would like to create an implementation of a refactoring tool for our prototype language instantiated with a real derivation system. From the refactorings we have looked at so far, we have noticed that many can be built from smaller, *atomic* refactorings. The *move object* refactoring can be built from repetitions of the simpler *swap object* refactoring. We would like to investigate this further, perhaps coming up with a refactoring calculus. Fowler, in [6], discusses *bad smells* in code that indicate that a refactoring would be desirable. Typical examples are *duplicated code* and *long method*. These translate nicely into *duplicated proof steps* and *long proofs*. We are interested in discovering more, proof-specific smells. One particular methodology would be to analyse the version history of large development under Subversion or CVS control.

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful suggestions. The first author was supported by Microsoft Research through its PhD Scholarship Programme.

References

1. D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3:309–330, 2010.
2. P. Corbineau. A declarative language for the Coq proof assistant. In *Types for Proofs and Programs*, LNCS. 2008.
3. M. Cornlio, A. Cavalcanti, and A. Sampaio. Refactoring by transformation. *Electronic Notes in Theoretical Computer Science*, 70(3):311 – 330, 2002.
4. E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 2006.
5. R. Ettinger and M. Verbaere. Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio. http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports, 2005.
6. M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.
7. G. Gonthier. The Four Colour Theorem: Engineering of a formal proof. *Computer Mathematics: 8th Asian Symposium, ASCM 2007*, pages 333–333, 2008.
8. T. C. Hales. Formal proof. *Notices of the AMS*, 55:1370–1380, 2008.
9. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on OSP*, pages 207–220. ACM, Oct 2009.
10. H. Li and S. Thompson. Formalisation of Haskell Refactorings. In *Trends in Functional Programming*, September 2005.
11. T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.
12. W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois, Champaign, IL, USA, 1992.
13. O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *User Interfaces for Theorem Provers, UITP*, 1998.
14. A. Schairer and D. Hutter. Proof transformations for evolutionary formal software development. In *Algebraic Methodology and Software Technology, LNCS*, volume 2422, pages 13–19. 2002.
15. A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. TACS '01, pages 535–559, 2001.
16. S. Stepney, F. Polack, and I. Toyn. Refactoring in maintenance and development of Z specifications. *Electr. Notes Theor. Comput. Sci.*, 70(3), 2002.
17. J. Urban and G. Bancerek. Presenting and explaining Mizar. *Electron. Notes Theor. Comput. Sci.*, 174(2):63–74, 2007.
18. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of TPHOLs '99*, pages 167–184, 1999.