An Essence of SSReflect

Iain Whiteside, David Aspinall, and Gudmund Grov

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

Abstract. SSReflect is a powerful language for proving theorems in the Coq system. It has been used for some of the largest proofs in formal mathematics thus far. However, although it constructs proofs in a formal system, like most other proof languages the semantics is informal making it difficult to reason about such proof scripts. We give a semantics to a subset of the language, using a hierarchical notion of proof tree, and show some simple transformations on proofs that preserve the semantics.

1 Introduction

The SSReflect language for Coq was initially developed by Gonthier to facilitate his proof style of *small scale reflection* during the pioneering formalisation of the Four Colour Theorem (FCT), [7, 9, 14]. SSReflect provides powerful matching facilities for precise rewriting, so-called *views* offering the power of reflection, and a host of language constructs for managing the large numbers of variables and assumptions typical of the combinatorial proofs encountered in the FCT. The language has matured and is now being used more widely in the Coq community [15, 13]. In particular, Gonthier and his team are using it in their formalisation of finite group theory and the Feit-Thompson theorem [8].

As in any programming or proof language, it is all too easy to write poorly structured – even unreadable – scripts during exploration and it is a tedious, often error-prone task to *refactor* proofs to make them presentable, to have good *style* [12]. Indeed, Gonthier claims to have spent months refactoring his proof of the FCT. The notion of "good" and "bad" style is cultural, and one reason for focussing on SSReflect is the clear ideas about what this style should be. Indeed, a large part of the language – the part that we are most keenly interested in – facilitates a style of proof that is designed to create proof scripts that are *robust*, *maintainable*, and *replayable*. Such proofs involves ensuring each line (or *sentence*) has a clear meaning mathematically (related formula manipulations, an inductive step, etc), keeping scripts as linear as possible, emphasising important goals, as well as supporting many convenient naming conventions.

The challenge is to provide refactorings that can be automated and, crucially, can be proved *correct*. That is, performing the refactoring will not break a proof. Unfortunately many existing proof languages, including SSReflect, have no formal semantics and their behaviour is determined by execution on goals, making

it difficult to relate equivalent proof scripts. In previous work [19], we investigated proof refactorings – as a structured and automatable way to transform a "bad" script into a "good" script – on a formally defined Isar-style declarative proof language [18]. Here we make a step in this direction for SSReflect by first modelling the semantics of (a subset of) the language, which we call *eSSence*, then demonstrating how some simple transformations may be rigorously defined. Our semantics is based on a tactic language called Hitac [1], which constructs hierarchical proof trees (or Hiproofs) when executed. Hiproofs [4] offer the advantage that one can view the proof tree at many levels of abstraction: hiding or showing as much detail as desired. We use Hiproofs as an underlying proof framework because the structured proofs written in SSReflect have a very natural hierarchical interpretation and give us novel ways to view the proof.

Contributions. We have identified the main contributions of our work as 1) providing a clear operational semantics for the eSSence language; and, 2) providing some initial refactorings of SSReflect proofs. Furthermore, we believe this presentation will help disseminate both the novel features of the language – which are by no means Coq-specific – and the Hiproof and Hitac formalisms.

Outline. In the next section, we introduce the eSSence language by example and briefly sketch the Hitac and Hiproof formalisms. Section 3 introduces a simple type theory as the underlying logic for our language. The syntax and semantics of eSSence is described in Sections 4 and 5. We then show how to use the semantics to refactor our example in Section 6, before concluding in Section 7.

2 Background

The simple eSSence (and SSReflect) script, shown below left, is a proof of $(A \to B \to C) \to (A \to B) \to A \to C$.

```
move => h1 h2 h3.
move : h1.
apply.
-by [].
-apply: h2.
 by [].
```

```
move => AiBiC AiB Atrue.
apply: AiBiC ; first by []
by apply: AiB.
```

An eSSence proof is a *paragraph* that, in this case, consists of three *sentences* and then two nested paragraphs (the first a single sentence long, the second two sentences). The hyphens marking the start of each paragraph are *annotations* to explicitly show where the proof has branched. Each sentence operates on the first goal in a stack and any subgoals resulting from the execution of the sentence are pushed back on the top of the stack. We explain each sentence:

1. The `move` tactic here plays an explanatory role (acting as the identity tactic). The work is done by ':' and '=>', which are actually tacticals to *pop* and *push*

variables and assumptions to/from the context. This first sentence extends the context to $h1 : (A \to B \to C)$, $h2 : (A \to B)$, and $h3 : A$ with the goal being simply C.
2. The sentence `move` : h1. will then *push* h1 back into the goal, transforming it to $(A \to B \to C) \to C$. Note that the context is unordered, we can push arbitrary context elements as long as the context stays well-formed.
3. In the first branch, `apply` attempts to match the conclusion of the goal with the conclusion of the first assumption – motivating the previous step – then breaks down the assumption as new subgoals - in this case, two: A and B.
4. The `by` tactical attempts to solve a goal with the supplied tactic (alongside default automation applied after). The [] means that the goal is trivial and is solved by the default automation – the assumption h3 is used to solve A.
5. The second branch requires further application using h2 in the first sentence and solving by assumption in the second. In this branch, however, `apply:`, behaves differently from `apply` as it takes arguments, the term to be applied. With a single argument, for example h2, this behaves as `move` : h2. followed by `apply`.

In this example, we see two main features of the language: clustering all bookkeeping operations into two tacticals (: and =>) and providing structure and robustness to scripts using indentation, annotation, and `by`. The language has many more constructs, but this gives a good flavour.

However, this isn't a very well-presented script. It is too verbose and the assumption names do not convey any information. We can apply structured changes to improve the proof. We can compress `move` : h1. and `apply`. into a single line, using the THEN tactical (**;**) `move`: h1 **;** `apply`. Furthermore, we can transform it to simply `apply:` h1 as `move` behaves as an identity tactic. Finally, we can merge the *obvious* steps into the same line and rename the hypotheses. To refactor `apply`. and -`by` []., we utilise the `first` tactical whose supplied tactic operates only on the first subgoal of a branch. The resulting script is displayed above right. Now, each sentence can be understood as an – albeit simple – mathematical step. In Section 6, we show how to give provably correct transformations that can achieve this refactoring.

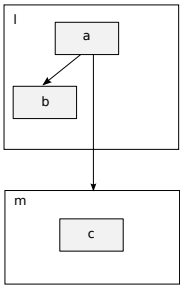

Fig. 1. A Hiproof

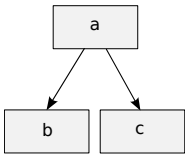

Fig. 2. The skeleton

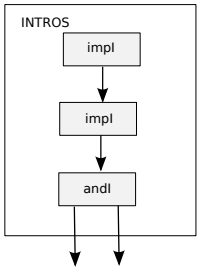

Fig. 3. INTROS

Hierarchical proof. As an underlying language for eSSence proofs, we use *Hiproofs*. First investigated by Denney et al in [4], 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. An abstract example of a Hiproof is given in Figure 1, where a, b, and c are called *atomic tactics* i.e. the inference rules of the language. Figure 1 reads as follows: the abstract tactic l first applies an atomic tactic a. The tactic a produces two subgoals; the first is solved 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. More concretely, Figure 3 shows the application of an *INTROS* tactic as a Hiproof. Following [1], we give a syntactic description of Hiproofs:

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

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. A *swap* switches the order of two goals. Labelling binds weakest, then sequencing with tensor binding most tightly. We can now give a syntactic description of the Hiproof in Figure 1: $([l]a \,;\, b \otimes id) \,;\, [m]\ c$. Atomic tactics, a, come from a fixed set $\mathcal{A}$ and what we call an *atomic tactic* is an inference rule schema:

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

stating that the atomic tactic a, when applied to the goal γ, breaks it down into subgoals $\gamma_1, \ldots, \gamma_n$. We say Hiproofs are defined in terms of a *derivation system*, which instantiates the *atomic goals* $\gamma \in \mathcal{G}$ and *atomic tactics* $a \in \mathcal{A}$. A Hiproof is *valid* if it is well-formed and if atomic tactics are applied correctly. Validation is defined as a relation on lists of goals $s \vdash g_1 \rightarrow g_2$, and is the key proof checking property. We expand on this notion of derivation system and provide an instantiation for eSSence in Section 3.

Hierarchical tactics. The Hitac language extends Hiproofs as follows:

$$t \;::=\; \ldots \;\mid\; assert\,\gamma \;\mid\; t \;\mid\; t \;\mid\; name(t, \ldots, t) \;\mid\; X$$

In addition to the standard Hiproof constructs, goal assertions ($assert\ \gamma$) can control the flow; alternation ($t \mid t$) allows choice; and, defined tactics ($name(t, \ldots, t)$) and variables (X) allow us to build recursive tactic programs. Tactic evaluation is defined relative to a *proof environment*, specifying the defined tactics available. Evaluation of a tactic is defined as a relation

$$\langle g, t\rangle \Downarrow^t_{\mathcal{E}} \langle s, g'\rangle,$$

which should be read as: 'the tactic t applied to the list of goals g returns a Hiproof s and remaining subgoals g', under the proof environment $\mathcal{E}$'. To

illustrate, we give the evaluation rules for *sequencing*, *tensor*, and *defined tactics* below, where we write $\overline{X}_n$ as shorthand for the variable list $[X_1, \ldots, X_n]$. For a full presentation see [1].

$$\frac{\langle g_1, t_1\rangle \Downarrow^t_{\mathcal{E}} \langle s_1, g\rangle \qquad \langle g, t_2\rangle \Downarrow^t_{\mathcal{E}} \langle s_2, g_2\rangle}{\langle g_1, t_1 \ ; \ t_2\rangle \Downarrow^t_{\mathcal{E}} \langle s_1 \ ; \ s_2, g_2\rangle} \quad \text{(B-Tac-Seq)}$$

$$\frac{\langle g_1, t_1\rangle \Downarrow^t_{\mathcal{E}} \langle s_1, g_1'\rangle \qquad \langle g_2, t_2\rangle \Downarrow^t_{\mathcal{E}} \langle s_2, g_2'\rangle}{\langle g_1 @ g_2, t_1 \otimes t_2\rangle \Downarrow^t_{\mathcal{E}} \langle s_1 \otimes s_2, g_1' @ g_2'\rangle} \quad \text{(B-Tac-Tens)}$$

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

The evaluation rules for tactics are correct and construct valid proofs:

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

With Hitac we can, for example, define a parameterised tactic $ALL(X)$, which applies the tactic supplied as a parameter to all subgoals, as follows:

$$\begin{array}{lcl} ALL(X) & := & X \otimes (ALL(X) \mid \langle\rangle) \\ ID & := & ALL(id) \end{array}$$

The supplied tactic, X, is applied to the first subgoal and the tactic is called recursively on the remaining goals; it will succeed and apply the empty tactic when the list of goals is empty. Thus, ID can be seen as a more general identity tactic. We can also define tactics which rotate and reflect the order of subgoals:

$$\begin{array}{ll} NULL & := \langle\rangle \mid id \\ ROTATE & := [\ (swap \otimes ID)\ ;\ (id \otimes (ROTATE) \ \mid \ \langle\rangle)\ ] \mid NULL \\ ROTATE\_R & := [\ (ID \otimes swap)\ ;\ (id \otimes (ROTATE\_R) \ \mid \ \langle\rangle)\ ] \mid NULL \\ REFLECT & := ROTATE\ ;\ ((REFLECT \otimes id) \mid \langle\rangle) \end{array}$$

The difference between $ROTATE$ and $ROTATE\_R$ is the direction of rotation. The tactic $ROTATE$ maps $[g_1, g_2, g_3, g_4]$ to $[g_2, g_3, g_4, g_1]$ and $ROTATE\_R$ maps to $[g_4, g_1, g_2, g_3]$. These are used to define the various rotation tacticals in SSReflect.

3 Underlying Logic of eSSence

In keeping with the origins of SSReflect, we instantiate the Hiproof framework with a type theory; however, we choose a less expressive theory than the Calculus of Inductive Constructions used in Coq to simplify the presentation but keep the flavour of the system. The logic is called λHOL and is well studied in e.g. Barendregt [5]. The set of types, $\mathcal{T}$, is given as follows:

$$\mathcal{T} ::= \mathcal{V} \ \mid \ \texttt{Prop} \ \mid \ \texttt{Type} \ \mid \ \texttt{Type}' \ \mid \ \mathcal{T}\,\mathcal{T} \ \mid \ \lambda\,\mathcal{V} : \mathcal{T}.\,\mathcal{T} \ \mid \ \Pi\,\mathcal{V} : \mathcal{T}.\mathcal{T}$$

where $\mathcal{V}$ is a collection of variables. A declaration is $x : A$ where $A \in \mathcal{T}$ and $x \in \mathcal{V}$. A *context* is a finite, ordered sequence of declarations, with all subjects distinct and the set of *sorts*, $s = \{\texttt{Prop}, \texttt{Type}, \texttt{Type}'\}$. Figure 4 enumerates the rules that axiomatise the notion of a typing judgement $\Gamma \vdash A : B$ (saying A has the type B in the context Γ). The pairs (s_1, s_2) are drawn from the set $\{(\texttt{Type}, \texttt{Type}), (\texttt{Type}, \texttt{Prop}), (\texttt{Prop}, \texttt{Prop})\}$, allowing construction of function types, universal quantification, and implication respectively.

Definition 1 (Atomic Goal) *A goal is a pair of a context, Γ and a type $P \in \mathcal{T}$, such that $\Gamma \vdash P : \texttt{Prop}$. We will write $\Gamma \vdash P$ for goals.*

$$\langle\rangle \vdash \texttt{Prop} : \texttt{Type} \qquad \langle\rangle \vdash \texttt{Type} : \texttt{Type}'$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \qquad \frac{\Gamma \vdash F : (\Pi x : A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}$$

Fig. 4. The typing rules for λHOL.

Atomic tactics, presented in Figure 5 as inference rules, should be read backwards: from a single goal, applying the rule gives zero or more subgoals. Side-conditions (restricting applicability) are also written above the line, but will be always the leftmost and not of a goal form.

A-Intro(n)**.** The *INTRO* tactic performs an introduction step. The subgoal generated is the obvious one and the assumption is given the name supplied.

A-Exact(t)**.** The *EXACT* tactic takes a term as a parameter and solves the goal if the term has a type convertible with the goal.

A-Refine(t)**.** Refinement takes a term of a convertible type to the current goal, containing proof variables – explicitly shown in the rule – and leaves the variables as subgoals. There is an additional side-condition on the rule A-Refine: that the variables x_i cannot occur inside the $P_1 \ldots P_n$.

Atomic tactics preserve *well-formedness* in the sense that given a well-formed goal (Definition 1), application of the tactic produces more well-formed goals. Another important property for a proof system is:

Conjecture 1 (Soundness of Atomic Tactics.) *The atomic tactics are sound with respect to the low-level rules of λHOL. That is, if we construct a proof term of the subgoals, then we can construct a proof term for the original goals.*

We have not proved this for our system, as it is not a key goal: rather the actual logical system serves to illustrate our approach.

$$\frac{\Gamma \vdash t : Q \qquad P =_\beta Q}{\Gamma \vdash P} \quad \text{(A-Exact}(t)\text{)}$$

$$\frac{(x : P) \in \Gamma \textit{ for some } x}{\Gamma \vdash P} \quad \text{(A-Assumption)}$$

$$\frac{n \notin \Gamma \qquad \Gamma, (n : T) \vdash U}{\Gamma \vdash \Pi x : T.U} \quad \text{(A-Intro}(n)\text{)}$$

$$\frac{x : T \in \Gamma \qquad wf(\Gamma \setminus x : T) \qquad \Gamma \setminus x : T \vdash \Pi x : T.P}{\Gamma \vdash P} \quad \text{(Revert}(x)\text{)}$$

$$\frac{\Gamma \vdash U : Prop \qquad \Gamma \vdash U \qquad \Gamma \vdash U \to P}{\Gamma \vdash P} \quad \text{(A-Assert}(U)\text{)}$$

$$\frac{t(?x_1 : P_1, \ldots, ?x_n : P_n) : Q \qquad P =_\beta Q \qquad \Gamma \vdash P_1 \ldots \Gamma \vdash P_n}{\Gamma \vdash P} \quad \text{(A-Refine}(t)\text{)}$$

Fig. 5. Atomic tactics

4 The eSSence Language

The syntax for eSSence is given in Figure 6. A *sentence* is a grammar element *sstac*. The parameters *num*, *term*, and *ident* stand for numerals, terms, and identifiers respectively. The basic tactics are `move`, `apply`, and one or more rewrite steps. The `have` tactic allows forward proof and `;` is the LCF THEN and THENL tacticals. The `by` tactical ensures that the supplied tactic solves the current goal. Using the `first` and `last` tacticals, one can operate on a subset of goals and `first` *n* `last` performs subgoal rotation. Most bookkeeping operations are performed using the *discharge* and *introduction* tacticals (`:` and `=>` respectively), which pop/push assumptions and variables from/to the context. The first *iitem* in an application of the introduction tactical can be a branching pattern, allowing for the case where the corresponding tactic introduces multiple subgoals, which we can deal with simultaneously. For a full presentation and tutorial guide to the original SSReflect language, see [9, 10].

Paragraphs. A proof script in SSReflect is simply a list of sentences, separated by full stops; however, one can optionally *annotate* a script with bullets ($*$, $+$, and $-$), which, along with indentation, helps make clear the subgoal flow within a script. The idea of these annotations is to use bullets to highlight where a proof branches. We build this directly into the eSSence language using the *sspara* grammar element. Paragraphs can be understood abstractly as a non-empty list of sentences followed by a possibly empty list of (indented) paragraphs. We add the restriction that in each paragraph the annotations must be the same bullets. There is no semantic difference between the various bullet symbols. The SSReflect annotation guidelines state that:

– If a tactic sentence evaluates and leaves one subgoal, then no indentation or annotation is required. If a tactic sentence introduces two subgoals – the `have` tactic, for instance – then the proof of the first goal is indented. The second is at the same level of indentation as the parent goal.

ssscript ::= *sspara*

sstac ::= *dtactic*
| `apply`
| `apply:`
| `rewrite` *rstep*$^+$
| `have:` *term* [`by` *sstac*]
| *sstac* `;` *chtac*
| `by` *chtac*
| `exact` *term*
| *sstac* `;` `first` [*num*] *chtacopt*
| *sstac* `;` `last` [*num*] *chtacopt*
| *sstac* `;` `first` [*num*] `last`
| *sstac* `;` `last` [*num*] `first`
| *dtactic*`:` *ditem* ... *ditem*
| *sstac*`=>` [*iitemstart*] *iitem* ... *iitem*

dtactic ::= `move`

chtac ::= *sstac* | `[`*sstac*`|`...`|`*sstac*`]`
chtacopt ::= *sstac* | `[`[*sstac*]`|`...`|`[*sstac*]`]`
iitemstart ::= *iitem* | `[`*iitem**`|`...`|`*iitem**`]`

ssanno ::= `+` | `-` | `*`

sspara ::= *sstac*`.`
⋮
sstac`.`
(*ssanno sspara*)*

rstep ::= ([`-`]*term*) | *sitem*

sitem ::= `/=` | `//` | `//=`

ditem ::= *term*

iitem ::= *sitem* | *ipattern*

ipattern ::= `ipatt` *ident*

Fig. 6. The eSSence language syntax

– If a sentence introduces three or more subgoals then bullets and indentation are required to mark the start of each subgoal's proof. The last, however, is outdented to the same level as the parent.

The outdenting of the final goal is to emphasise that it is somehow more difficult or interesting; this concept motivates the tacticals for rotating subgoals. We simplify the annotations by using explicit bullets even for the case of two subgoals and also bulleting the final subgoal. This notion of structuring corresponds exactly with the hierarchy in Hiproofs - every bullet corresponds to a labelled box. Figure 7 contains examples of scripts that follow our simplified structuring guidelines (on the right is an indication of the arity of each tactic). Our semantics, given next, enforces these guidelines and in Section 6 we give a syntax and semantics for scripts without annotation and describe how to annotate a script automatically.

5 Giving Meaning to eSSence Scripts

The semantics for eSSence is based on a static translation from eSSence to Hitac, written $[\![sstac]\!]$, giving us:

		$s1.$	$[\gamma] \to [\gamma_1, \gamma_2, \gamma_3]$	$s1.$	$[\gamma] \to [\gamma_1, \gamma_2]$
$s1.$	$[\gamma_1] \to [\gamma_2]$	$-\ s2.$	$[\gamma_1] \to []$	$-\ s2.$	$[\gamma_1] \to []$
$s2.$	$[\gamma_2] \to [\gamma_3]$	$-\ s3.$	$[\gamma_2] \to []$	$-\ s3.$	$[\gamma_2] \to [\gamma_3, \gamma_4]$
$s3.$	$[\gamma_3] \to []$	$-\ s4.$	$[\gamma_3] \to [\gamma_4]$	$+\ s4.$	$[\gamma_3] \to []$
		$s5.$	$[\gamma_4] \to []$	$+\ s5.$	$[\gamma_4] \to []$

Fig. 7. A linear script, one level of branching, and multiple branching levels.

$$\frac{\langle \gamma, [\![sstac]\!] \rangle \Downarrow^t_{\mathcal{E}} \langle s, g \rangle}{\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle}$$

which says that under an *environment* $\mathcal{E}$ the tactic *sstac* applied to the goal γ results in a list of generated subgoals g and a Hiproof s if the translated Hitac tactic behaves the same. We present most of the translation rules in Figures 8 and 13. We will explain the rules in Figure 8 and leave the rest for the reader.

$$
\begin{array}{lcl}
[\![\texttt{move}]\!] & = & [move]\ assert\ (\Gamma \vdash \Pi x : A.B)\ \mid\ HNF \\
[\![\texttt{by}\ sstac]\!] & = & [by]\ [\![sstac]\!]\ ;\ ALL(DONE)\ ;\ \langle\rangle \\
[\![sstac_1\ \texttt{;}\ \texttt{first}\ sstac_2]\!] & = & [\![sstac_1]\!]\ ;\ ([first]\ [\![sstac_2]\!]) \otimes ID \\
[\![sstac\ \texttt{;}\ \texttt{first last}]\!] & = & [FL]\ [\![sstac]\!]\ ;\ REFLECT \\
[\![sstac\ \texttt{;}\ \texttt{first}\ k\ \texttt{last}]\!] & = & [FkL]\ [\![sstac]\!]\ ;\ ROTATE;^k \\
[\![sstac\texttt{=>}\ iitem_1 \ldots iitem_n]\!] & = & [=>]\ [\![sstac]\!]\ ;\ [\![\texttt{ipat}\ iitem_1]\!]\ ;\ \ldots\ ;\ [\![\texttt{ipat}\ iitem_n]\!] \\
[\![\texttt{ipat}\ ident]\!] & = & INTRO(id)
\end{array}
$$

Fig. 8. eSSence evaluation semantics part one

$[\![\texttt{move}]\!]$**.** The `move` tactic behaves as an identity if an introduction step is possible or transforms the goal to head normal form otherwise. We model this by providing an assertion for checking a product and applying a tactic *HNF*, which reduces the goal to head normal form. Note also that we 'box up' the application of `move` to abstract away from any normalisation done by this tactic. This is the first of many occasions in the semantics where we use hierarchy to hide away details.

$[\![\texttt{by}\ sstac]\!]$**.** This rule evaluates the primitive form of the closing tactical, applying the DONE tactic to all subgoals after its parameter as some sort of default automation (including A-Assumption). We assume it to be built

from more primitive tactics. The empty hiproof is used to fail the *by* tactical if the goals are not solved after this automation is applied. The translation is invoked recursively on *sstac*.

$[\![sstac_1$ `;` `first` $sstac_2]\!]$**.** In this primitive form of selection tactical, we apply $sstac_2$ to only the first subgoal generated and add hierarchy to hide the proof.

$[\![sstac$ `;` `first last`$]\!]$**.** This tactical simply reflects the subgoals and is implemented by the Hitac reflection tactic described earlier.

$[\![sstac$ `;` `first` k `last`$]\!]$**.** To the subgoals $[g_1, g_2, g_3, g_4, g_5]$, $sstac_1$ `;` `first` 2 `last` would result in the remaining goals looking like $[g_3, g_4, g_5, g_1, g_2]$ and it is implemented by an appropriate number of rotations. The syntax $t\ ;^n$ simply means a sequence of applications of a tactic t of length n i.e. $t\ ;\ldots;\ t$.

$[\![sstac$`=>` $iitem_1 \ldots iitem_n]\!]$ **and** $[\![$`ipat` $ident]\!]$**.** These rules are used to translate the non-branching version of the introduction tactical, an instance of which would be $sstac$`=>` $iitem_1 \ldots iitem_n$. The tactic *sstac* is evaluated first; then each $iitem_i$ left to right. Each *iitem* can be either a simplification item or an *ipattern* and each *ipattern* is simply an introduction step.

Paragraphs. We represent scripts abstractly as a pair of lists: a *sstac* list and a paragraph list and we extend the evaluation relation to operate on a list of goals and produce a list of subgoals and a Hiproof. The evaluation rules follow:

$$\frac{\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, []\rangle}{\langle [\gamma], ([sstac], [])\rangle \Downarrow_{\mathcal{E}} \langle [S]s, []\rangle} \quad \text{(SS-Tac)}$$

$$\frac{sstacs \neq [] \qquad \langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, [\gamma']\rangle \qquad \langle [\gamma'], (sstacs, ssparas)\rangle \Downarrow_{\mathcal{E}} \langle s, []\rangle}{\langle [\gamma], (sstac :: sstacs, ssparas)\rangle \Downarrow_{\mathcal{E}} \langle ([S]s_\gamma); s, []\rangle} \quad \text{(SS-TacCons)}$$

$$\frac{\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, g\rangle \qquad length(g) > 1 \qquad \langle g, ([], ssparas)\rangle \Downarrow_{\mathcal{E}} \langle s, []\rangle}{\langle [\gamma], ([sstac], ssparas)\rangle \Downarrow_{\mathcal{E}} \langle ([S]s_\gamma); s, []\rangle} \quad \text{(SS-ParaStart)}$$

$$\frac{\langle [\gamma], sspara\rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, []\rangle \qquad \langle g, ([], ssparas)\rangle \Downarrow_{\mathcal{E}} \langle s_g, []\rangle}{\langle \gamma :: g, ([], sspara :: ssparas)\rangle \Downarrow_{\mathcal{E}} \langle ([P]s_\gamma) \otimes s_g, []\rangle} \quad \text{(SS-ParaCons)}$$

$$\langle [], ([], [])\rangle \Downarrow_{\mathcal{E}} \langle \langle\rangle, []\rangle \quad \text{(SS-ParaEnd)}$$

The idea is that given a paragraph (a pair of sentence and sub-paragraph lists), we first evaluate each sentence sequentially. All sentences except the last must return one goal (SS-TacCons). The last must either solve the goal and be the end of the paragraph (SS-Tac) or leave $n > 1$ and contain n nested paragraphs (SS-ParaStart). We apply each paragraph to each goal, then label and glue the proofs together (SS-ParaCons). Each sentence is also labeled to make clear the script structure. In this, we simply labelled each sentence and paragraph with S or P; however, in future we plan to allow supplied names. It can be shown that the evaluation relation behaves suitably:

Proposition 1 (Correctness of Evaluation) *If* $(\Gamma \vdash \quad P) \equiv \gamma$ *is a well-formed proposition, and* $\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle$ *then* s *is* ***valid****. That is,* $s \vdash [\gamma] \rightarrow g$.

Example. Recall our proof from Section 2, shown here with sentences labelled:

```
move => AiBiC  AiB  Atrue.      sent1
apply : AiBiC ; first by []     sent2
by apply : AiB.                 sent3
```

This script is parsed into a paragraph consisting of three sentences (and no additional paragraphs). At the script level, each sentence is evaluated sequentially using the rule SS-TacCons twice and then SS-Tac to finish the proof (since it operates on a singleton sentence list and empty paragraph list). This gives us the high-level Hiproof in Figure 9. At the level of sentences, we have:

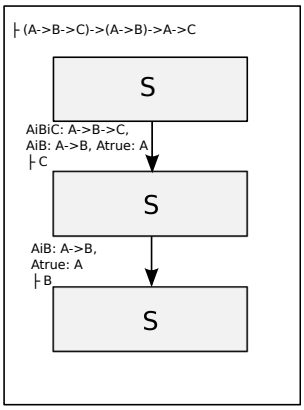

Fig. 9. High level view of Hiproof

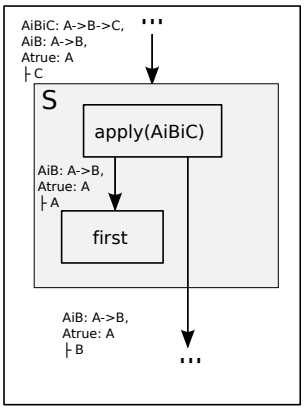

Fig. 10. The Hiproof for *sent2*

- *sent1* is translated by applying introduction tactical translation rule at the top level. During tactic evaluation, `move` behaves like an identity then each *iitem* applies the INTRO tactic to generate a goal:

$$AiBiC : A \to B \to C,\ \ AiB : A \to B,\ \ Atrue : A \vdash C.$$

- *sent2*, whose root is an application of the `first` tactical (where $sstac_1$ is `apply` : $AiBiC$ and $sstac_2$ is `by` []), is then translated. The resulting tactic calls `apply` and generates *two subgoals*: A and B in a context with AiB and $Atrue$. The second part of the translated `first` tactical – corresponding to the closing tactical `by` [] – is evaluated and solves the goal.
- The final sentence is used to solve the goal $AiB : A \to B,\ \ Atrue : A \vdash B.$ and proceeds similarly to *sent2*.

Figure 10 shows one *view* of the Hiproof generated by execution of *sent2*, in context with the rest of the proof. Here we see a little of the power of Hiproofs, as we can effectively hide the details of the proof of the first subgoal.

6 Refactoring eSSence

We are now in a position to demonstrate a small number of refactorings and illustrate the general approach to showing correctness. In order to do so we first introduce a notion of *unstructured script*, which is simply a list of sentences (*sstac*s). From an eSSence script, we can easily obtain an unstructured script by simply dropping annotations, and collapsing paragraphs into a single list. We end up, for example, using the second example in Figure 7, with the script represented as $s_1 :: s_2 :: \ldots s_n :: []$ (or $[s_1, \ldots, s_n]$), given in Figure 11.

$s1.\ [\gamma] \rightarrow [\gamma_1, \gamma_2, \gamma_3]$
$s2.\ [\gamma_1] \rightarrow []$
$s3.\ [\gamma_2] \rightarrow []$
$s4.\ [\gamma_3] \rightarrow [\gamma_4]$
$s5.\ [\gamma_4] \rightarrow []$

Fig. 11. Unstructured script.

$$\langle [], [] \rangle \Downarrow_{\mathcal{E}} \langle \langle \rangle, [] \rangle \quad \text{(NS-Emp)}$$

$$\frac{\langle \gamma, ss \rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, g_\gamma \rangle \qquad n = len(g) - 1 \qquad \langle g_\gamma \mathbin{@} gs, tacs \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle}{\langle \gamma :: gs, ss :: tacs \rangle \Downarrow_{\mathcal{E}} \langle (s_\gamma \otimes id^n)\ ;\ s, g \rangle} \quad \text{(NS-Cons)}$$

Fig. 12. Evaluation of unstructured script.

We give a semantics to evaluation with Figure 12 (writing id^n for an identity tensor of length n i.e. $id \otimes \ldots \otimes id$). The rule NS-Cons peels off a goal from the stack and applies the first tactic to it; it then pushes the resulting subgoals onto the stack (pre-appending to a list) and recurses with the rest of the tactics to be applied. The Hiproof is pieced together by tensoring together identity tactics to 'skip' the rest of the goals in the initial list, with NS-Emp dealing with the empty list case. Crucially, if an unstructured script is well-formed and evaluates successfully, we can introduce structure using an *annotation* operation. Annotation requires only information about the *arity* of each tactic (that is, the number of resulting subgoals). Annotation and flattening are dual operations and we have the following important property:

Theorem 2 (Correctness of annotation) *If we have an unstructured script, sstacs, such that:* $\langle \gamma, sstacs \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$*, then:* $\langle \gamma, annotate(sstacs) \rangle \Downarrow_{\mathcal{E}} \langle s', g \rangle$.

This means, that if we wish we can ignore the structure when reasoning about refactorings as it doesn't affect the resulting subgoals, just the Hiproof.

Transforming our example. In Section 2, we modified a simple proof script to improve its style. In particular, the specific operations we used:

- **Rename hypothesis:** to rename, for example, $h1$ to $AiBiC$.
- **Merge sentences:** to shorten the proof by collapsing to sentences into one.
- **Propogate closing tactical:** to move a `by` tactical outwards.
- **Replace `move` instance:** replacing it with `apply:`.

We focus only on *propagate* and *merge* in this paper.

Propagate. This refactoring moves an instance of by outside another tactical, if possible. There is a transformation rule for each suitable tactical, for example:

$$\frac{}{propagate(sstac_1 \ ; \ \texttt{by} \ sstac_2) \longrightarrow \texttt{by} \ sstac_1 \ ; \ sstac_2} \quad \text{(PROP-1)}$$

Propagate is correct if the sentence evaluates as $\langle \gamma, sstac_1 \ ; \ \texttt{by} \ sstac_2 \rangle \Downarrow_{\mathcal{E}} \langle s, [] \rangle$ and then $\langle \gamma, propagate(sstac_1 \ ; \ \texttt{by} \ sstac_2) \rangle \Downarrow_{\mathcal{E}} \langle s', [] \rangle$, for some s'. The condition that the original sentence must successfully evaluate is called a *precondition* and is important as often refactorings are not universally applicable. The semantics of the *closing tactical* by *sstac* relates it directly to the Hitac tactic *sstac* ; $ALL(DONE)$; $\langle\rangle$. This means the LHS of PROP-1 is $sstac_1$; $ALL(sstac_2$; $ALL(DONE)$; $\langle\rangle)$, which is equivalent to

$$(sstac_1 \ ; \ ALL(sstac_2)) \ ; \ ALL(DONE) \ ; \ \langle\rangle \equiv \texttt{by} \ sstac_1 \ ; \ sstac_2.$$

We can perform this refactoring inside a larger proof – here we only specified it on a single line – by integrating it with another refactoring called *transform sentence*, which has the obvious behaviour and is easily shown correct for unstructured scripts by a standard structural induction.

Merge sentences. There are two variations of this refactoring:

1. We merge `move` : h1. and `apply` into a single line, using the THEN tactical, since `move` : h1. generates a single subgoal.
2. Since `apply`: h1 generates two subgoals and - by [] is the first, use the `first` tactical to merge them as `apply`: h1 ; `first` by [].

The first version first drops annotations and works on unstructured scripts by stepping through the sentence list until the sentences to be merged are encountered. Then the following rule is applied:

$$\frac{arity(s_1) = 1}{merge(s_1, s_2, s_1 :: s_2 :: sstacs) \longrightarrow s_1 \ ; \ s_2 :: sstacs} \quad \text{(SS-MERGE-1)}$$

The second case requires more sophistication. If $arity(s_1) > 1$, then we can merge it with the *first* sentence of any of the subparagraphs, using the `first` tactical. The transformation rule can be seen on structured scripts as the following, where s is the first sentence, and i is the position of the second sentence in the paragraph list. It merges the first sentence in the appropriate paragraph $(head(fst(sspara_i)))$ with the supplied sentence.

$$\frac{\begin{array}{c} arity(s) = n \qquad sspara_i' = (tl(fst(sspara_i)), snd(sspara_i)) \qquad sstac = hd(fst(sspara_i)) \\ ssparas = [sspara_1, \ldots, sspara_{i-1}, sspara_i', sspara_{i+1}, \ldots, sspara_n] \end{array}}{merge(s, i, ([s], [sspara_1, \ldots, sspara_n])) \longrightarrow ([s \ ; \ first \ i \ [sstac] \ ], ssparas)} \quad \text{(SS-MERGE-2)}$$

Correctness of the first version is again an easy induction on unstructured scripts; however, the second case is more difficult as it involves manipulating an

arbitrary paragraph list. We identify the precise paragraph to refactor and use the *transform paragraph* refactoring to simplify the problem. We can then induct on the transformation over structured scripts and use equivalences in the Hitac language to show that evaluation succeeds.

7 Conclusions

In this work, we have identified and provided a semantics for an important subset of the SSReflect language, dealing primarily with proof style. Furthermore, we believe we have disseminated the principles of the language in a clearer setting and taken advantage of its inherent hierarchical nature by basing our semantics on the Hiproof framework. Importantly for our future program of work, this semantics enables us to reason about SSReflect scripts and, in particular, justify the correctness of refactorings.

Related work. Most related to our work are [17, 11], where a formal semantics is given for a declarative language and a procedural language, which is used to recover proof scripts from the underlying proof term. There has also been some work on formal semantics for proof languages, such as C-zar for Coq, and the Ωmega proof language [3, 2], but we are not aware of any work which specifically attempts to take advantage of this for these languages.

Refactoring is well-studied for programming languages and the literature is well surveyed by Mens et al [16]. The canonical reference for programming language refactoring is Fowler [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, and many of these refactorings in the domain of programming map across directly to proof.

Further work. There are many ways to progress this work. In particular, we would like to extend our semantics to cover a larger set of the SSReflect language; in particular, we would like to study *clear switches* in SSReflect, which allow the user to delete assumptions from the context. These are an important part of the language's design and make data flow explicit. In this work, we have not dealt with meta-variables. This is something that we would like to investigate, to see if our techniques could scale to a language which formally deals with meta-variables. With the refactorings described above, we have only scratched the surface of what is possible and plan to focus both on refactorings that are specific to the SSReflect language and translating refactorings that we have studied in previous work into a new language [19].

Acknowledgements. The authors would like to thank Georges Gonthier for the many useful discussions which motivated this work. The first author was supported by Microsoft Research through its PhD Scholarship Programme. The third author was supported by EPSRC grant EP/H024204/1.

References

1. D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3:309–330, 2010.
2. S. Autexier and D. Dietrich. A tactic language for declarative proofs. In *ITP*, pages 99–114, 2010.
3. P. Corbineau. A declarative language for the Coq proof assistant. In *Types for proofs and programs*, TYPES'07, pages 69–84. Springer-Verlag, 2008.
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. H. B. et al. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.
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. G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA, 2007.
9. G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.
10. G. Gonthier and R. Stéphane Le. An Ssreflect Tutorial. Technical Report RT-0367, INRIA, 2009.
11. F. Guidi. Procedural representation of cic proof terms. *J. Autom. Reason.*, 44(1-2):53–78, Feb. 2010.
12. J. Harrison. Proof style. In *TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer-Verlag.
13. J. Heras, M. Poza, M. Dns, and L. Rideau. Incidence simplicial matrices formalized in Coq/SSReflect. In *Intelligent Computer Mathematics*, volume 6824 of *LNCS*, pages 30–44. Springer Berlin / Heidelberg, 2011.
14. G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant: A tutorial. August 2007.
15. V. Komendantsky. Reflexive toolbox for regular expression matching: verification of functional programs in Coq+SSReflect. PLPV '12, pages 61–70, 2012.
16. T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.
17. C. Sacerdoti Coen. Declarative representation of proof terms. *J. Autom. Reason.*, 44(1-2):25–52, Feb. 2010.
18. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of TPHOLs '99*, pages 167–184, 1999.
19. I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. MKM'11, pages 260–275, Berlin, Heidelberg, 2011. Springer-Verlag.

$$
\begin{array}{lcl}
[\![\texttt{apply}]\!] & = & [apply]\ INTRO(top)\ ; \\
 & & (REFINE(top) \mid REFINE(top\ \_) \mid \ldots) \\
[\![\texttt{apply:}\ t_1 \ldots t_n]\!] & = & [apply]\ (REFINE(t_1(t_2 \ldots t_n)) \mid REFINE(t_1\ \_\ (t_2 \ldots t_n)) \mid \ldots) \\
[\![\texttt{rstep}\ rev\ tm]\!] & = & [rstep]\ REWRITE(rev, tm) \\
[\![\texttt{rewrite}\ rstep_1 \ldots rstep_n]\!] & = & [rewrite]\ [\![rstep_1]\!]\ ;\ \ldots\ ;\ [\![rstep_n]\!] \\
[\![\texttt{exact}\ term]\!] & = & [exact]\ EXACT(tm) \\
[\![\texttt{by}\ [sstac_1 | \ldots | sstac_n]]\!] & = & [by]\ [\![sstac_1]\!]\ ;\ ALL(DONE)\ ;\ \langle\rangle \mid \ldots \mid \\
 & & \quad [\![sstac_n]\!]\ ;\ ALL(DONE)\ ;\ \langle\rangle \\
[\![\texttt{have:}\ term\ \texttt{by}\ sstac]\!] & = & ASSERT(tm)\ ;\ ([\![sstac]\!] \otimes id) \\
[\![sstac_1\ \texttt{;}\ sstac_2]\!] & = & [\![sstac_1]\!]\ ;\ ALL(\ [\![sstac_2]\!]) \\
[\![sstac_0\ \texttt{;}\ [sstac_1 | \ldots | sstac_n]]\!] & = & [\![sstac_0]\!]\ ;\ ([\![sstac_1]\!] \otimes \ldots \otimes [\![sstac_n]\!]) \\
[\![sstac_1\ \texttt{;}\ \texttt{last}\ sstac_2]\!] & = & [\![sstac_1]\!]\ ;\ ID \otimes ([last]\ [\![sstac_2]\!]) \\
[\![sstac_0\ \texttt{;}\ \texttt{last}\ k\ [sstac_1 | \ldots | sstac_n]]\!] & = & [\![sstac_0]\!]\ ;\ ID\ \otimes ([last][\![sstac_1]\!]) \otimes \ldots \otimes ([last][\![sstac_n]\!]) \otimes id_{k-1} \\
[\![sstac_0\ \texttt{;}\ \texttt{first}\ k\ [sstac_1 | \ldots | sstac_n]]\!] & = & [\![sstac_0]\!]\ ;\ id_{k-1}\ \otimes ([first][\![sstac_1]\!]) \otimes \ldots \otimes ([first][\![sstac_n]\!]) \otimes ID \\
[\![sstac\ \texttt{;}\ \texttt{last first}]\!] & = & [LF]\ [\![sstac]\!]\ ;\ REFLECT \\
[\![sstac\ \texttt{;}\ \texttt{last}\ k\ \texttt{first}]\!] & = & [LkF]\ [\![sstac]\!]\ ;\ ROTATE\_R;^k \\
[\![\texttt{ditem}\ term]\!] & = & REVERT(tm) \\
[\![dtactic\texttt{:}\ ditem_1 \ldots ditem_n]\!] & = & [:]\ [\![\texttt{ditem}\ term_n]\!]\ ;\ \ldots\ ;\ [\![\texttt{ditem}\ term_1]\!]\ ;\ [\![dtactic]\!] \\
[\![//]\!] & = & ALL([DONE]DONE) \\
[\![/=]\!] & = & ALL([SIMP]SIMP) \\
[\![//=]\!] & = & [\![/=]\!]\ ;\ [\![//]\!]
\end{array}
$$

Fig. 13. Remaining translation rules