REFACTORING PROOFS

IAIN JOHNSTON WHITESIDE

Doctor of Philosophy
School of Informatics
University of Edinburgh
2013

ABSTRACT

Refactoring is an important Software Engineering technique for improving the structure of a program after it has been written. Refactorings improve the maintainability, readability, and design of a program without affecting its external behaviour. In analogy, this thesis introduces *proof refactoring* to make structured, semantics preserving changes to the proof documents constructed by interactive theorem provers as part of a formal proof development.

In order to formally study proof refactoring, the first part of this thesis constructs a *proof language framework*, Hiscript. The Hiscript framework consists of a procedural tactic language, a declarative proof language, and a modular theory language. Each level of this framework is equipped with a formal semantics based on a hierarchical notion of proof trees. Furthermore, this framework is generic as it does not prescribe an underlying logical kernel. This part contributes an investigation of semantics for formal proof documents, which is proved to construct valid proofs. Moreover, in analogy with type-checking, static well-formedness checks of proof documents are separated from evaluation of the proof. Furthermore, a subset of the SSReflect language for Coq, called eSSence, is also encoded using hierarchical proofs. Both Hiscript and eSSence are shown to have language elements with a natural hierarchical representation.

In the second part, proof refactoring is put on a formal footing with a definition using the Hiscript framework. Over thirty refactorings are formally specified and proved to preserve the semantics in a precise way for the Hiscript language, including traditional structural refactorings, such as *rename item*, and proof specific refactorings such as *backwards proof to forwards proof* and *declarative to procedural*. Finally, a concrete, generic refactoring framework, called Polar, is introduced. Polar is based on graph rewriting and has been implemented with over ten refactorings and for two proof languages, including Hiscript.

Finally, the third part concludes with some wishes for the future.

ACKNOWLEDGEMENTS

Writing this thesis has been the final part of the wonderful adventure that has been my PhD. Like all adventurers, I've needed a lot of help and support along the way. I'd like to take this opportunity to say thanks.

First and foremost I am deeply indebted to my supervisors David Aspinall, Lucas Dixon, and Gudmund Grov enough for their guidance, enthusiasm, and patient explanation when I so often needed it. When Lucas moved across the Atlantic to work for Google — I was a very difficult student — Gudmund was kind enough to step into the breach as my second supervisor. A decision perhaps regretted by Gudmund but much appreciated by me. This thesis has benefitted considerably from their careful proof reading, which has eliminated as much of my writing 'style' as possible. I'd like to also thank Georges Gonthier and Microsoft Research for sponsoring this work. Georges' unrivalled experience in formal proof and his enthusiasm for refactoring have been a great source of inspiration.

My fellow DReaMers helped create a wonderful environment for research and for providing me with an understanding of the clarity of mind, rigour, and objectivity that are crucial for doing science. Thank you.

Dominik, you instilled in me a love of parsing and I am profoundly thankful to have had a chance to work with you. I'm sorry we never did have that table tennis match, but let's face it, you would have beaten me and I don't like losing.

I am enormously grateful to Ewen Denney for giving me the dream opportunity of working with the Intelligent Systems Division at NASA Ames Research Center in sunny California. Aside from the stimulating chats and walks with Ewen and the enlightening conversations over burritos, the discovery of Anchor Steam, Sierra Nevada, sweet potato fries, and running in Yosemite with Matthias and David remain especially vibrant in my mind.

Outside of research, I must thank all my running friends from the Haries and Carnethy. I've had too many memorable experiences with you all to count. Running keeps me sane — approx. 1150 miles ran writing this — and I've been privileged to have such wonderful people to run with. Football thursday was a fantastic way to spend one afternoon from the working week with great friends. I will miss the fun, though perhaps not the barbarity of the 'cage' more than I can say.

To my flatmates and officemates: thank you for putting up with me and sorry about all the muddy shoes.

Last but by no means least, I'd like to thank my family whose unconditional faith and encouragement have kept me going. My mum is used to taking the brunt of my frustrations, so this is the perfect place to say thank you. For everything.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Edinburgh, 2013

Iain Johnston Whiteside,
September 11, 2017

CONTENTS

1

PRELIMINARIES

1.1 THE ORIGIN OF THE THESIS

The research presented in this dissertation has its origins in the observation that the overall *process* of interactive theorem proving is very much like that of programming. However, while software engineers have a wide variety of tools at their disposal, budding proof engineers have had to 'make do' with basic environments that are akin to those used for programming in the 80's. Over thirty years of research into Software Engineering (SE) has resulted in a wide variety of tools and techniques to support the software life-cycle. Large proof developments have a similar life-cycle, but it is not yet well-supported. Unfortunately, it is not just a simple matter of taking off-the-shelf SE tools. New research is required to develop the tools of the trade for *Proof Engineering*.

Within this grand vision, this thesis takes a modest first step by attempting to adapt the popular technique of *refactoring*. This aim is ambitious: there is over twenty years of research to catch up on, in a field that is still active and where tool support is at an early stage. Moreover, proofs and programs have important differences, many of which have an impact on proof refactoring research.

Early on, a question became apparent:

> What *is* a proof refactoring?

For programming, this question has a time-honoured, straightforward answer, well summarised by Opdyke:

> A semantics preserving restructuring operation 'that support[s] the design, evolution and reuse [of software]' (Opdyke, 1992).

Taking this definition, it is not clear what *semantics preservation* means for a proof refactoring. In pursuit of this and a more foundational understanding of proof developments, I decided upon a research program to construct a *minimal proof framework* in which to investigate refactorings. This framework should consist of a clean underlying notion of formal proof, a procedural and a declarative proof language, and a theory infrastructure. Each 'level' of the framework should have a clearly defined semantics. At the lowest level, I use hiproofs (Denney et al., 2006). Apart from being a suitable representation of underlying proof, the Hiproof formalism[1] has two additional novelties that captured my interest: it facilitates a hierarchical representation

1 Throughout this thesis, I use the convention that I write a capitalised Hiproof to refer to the language and proofs in the language with a lowercase hiproofs.

of proofs and could be defined generically, without recourse to a particular logical language. On top of this proof representation, I develop two proof languages: Hiscript and eSSence, and I extend Hiscript to a theory language, completing the proof language framework.

This thesis contains over thirty formally specified refactorings for the Hiscript language, each of which has a proof that it preserves the underlying semantics of the language in a precisely defined way. Finally, I describe a graph-based approach for proof refactoring that I developed and implemented in joint work with Dominik Dietrich. POLAR, our PrOof LAnguage Refactoring framework provides a generic mechanism for refactoring proof languages and a prototype implementation supports two proof languages.

1.2 THESIS ROADMAP

As a roadmap to this thesis, I present and describe the content of each chapter. If the chapter contains original contributions, these are enumerated.

This thesis consists of three parts and an appendix. The first part deals exclusively with the proof language framework and its semantics. The second is dedicated to proof refactoring. Each part also has its own introduction chapter. The final part concludes the thesis and suggests future work.

PART 1: PROOF LANGUAGE FRAMEWORK

CHAPTER 2: PROOF LANGUAGES AND HIERARCHY.

This chapter motivates and describes the approach to proof language semantics in Part 1. It sketches the relevant background on hierarchical proof and proof languages.

CHAPTER 3: HIPROOFS AND HITACS.

This chapter describes the foundations of the Hiscript language framework. It introduces the syntax and semantics for hiproofs and *Hitac*, a hierarchical tactic language on which Hiscript and eSSence are based.

While this chapter contains mostly background material, it contains some original contributions:

1. The Hiproof and Hitac languages are extended with constructs used in this thesis.
2. A notion of normal form for hiproofs is provided.
3. *Well-formedness checking* for hitacs is introduced to provide a mechanism that detects some simple causes of failure in a static way.

CHAPTER 4: THE HISCRIPT PROOF LANGUAGE

This chapter extends the framework by introducing a declarative proof language called Hiscript. I provide a formal semantics for Hiscript and prove some theoretical properties. The original contributions of this chapter are:

1. The proof language Hiscript is *generic*: it does not prescribe an underlying logic; and it is hierarchical: providing explicit (and also implicit) hierarchical constructs.
2. The evaluation semantics for Hiscript is proved correct in the technical sense that it constructs valid hiproofs.
3. Well-formedness checking rules for Hiscript are shown to trap potential errors before evaluation.
4. A precise notion of *gaps* in Hiscript proofs is given.

CHAPTER 5: HISCRIPT THEORIES.

This chapter introduces extends Hiscript with a language for constructing collections of lemmas called *theories*. I give a precise evaluation semantics for theories, based on the linear ML-style model of evaluation where evaluating a theory will construct a *proof environment*. A proof environment is an abstract representation of the items in a theory. I then introduce a simple theory import mechanism for collections of theories, called *proof documents*. The original contributions of this chapter are:

1. A formal semantics for theories that separates static *well-formedness* checks from *proof checking*.
2. A formal semantics for a simple inheritance mechanism for theories that also includes an export interface mechanism using the *public/private* notion known from object-oriented programming.
3. The semantics are proved to construct *well-formed* proof environments.

CHAPTER 6: THE ESSENCE LANGUAGE.

This chapter, independent of the previous two, presents the eSSence language, which is a subset of SSReflect. I define the language semantics in terms of Hitac and prove that the evaluation semantics is correct. The original contributions of this chapter are:

1. The eSSence language is presented in a formal way.
2. Two formal semantics are given: a direct translation to Hitac and a big step evaluation relation. Furthermore, these two relations are proved to coincide.
3. The hierarchy in hiproofs is demonstrated to have a natural mapping to the annotation language of SSReflect and provides structure to the underlying proof.

PART 2: PROOF REFACTORING

CHAPTER 7: IMPROVING THE DESIGN OF EXISTING CODE.

This chapter introduces refactoring: first in its original incarnation with programming languages, then the analogous concept of *proof refactoring*. I pay attention to the important concept of semantics preservation and survey some of the main techniques for programming language refactoring.

CHAPTER 8: A CATALOGUE OF PROOF REFACTORINGS.

This chapter further introduces proof refactoring by presenting eight examples of refactoring in an informal way. Each refactoring is described alongside some motivation for performing it. I also give a step-by-step recipe for performing it by hand and an example of each refactoring. The style of these refactorings is modelled on Martin Fowler's refactoring "bible" (Fowler, 1999). The original contribution of this chapter is:

1. This chapter contributes towards the body of knowledge of refactoring by enumerating a small set of refactorings for proof documents.

CHAPTER 9 AND 10: REFACTORING PROOFS AND REFACTORING THEORIES.

This chapter and the next utilise the formal semantics for Hiscript, introduced in Part 1, to give formal specifications for over 20 refactorings. Furthermore, I use the semantics to prove a correctness property: that these refactorings are *semantics preserving* in a precise sense that I define. The original contributions of these chapters are:

1. Formal definitions for proof refactoring and semantics preservation are given for Hiscript.
2. A collection of formally specified, correct refactorings are given for Hiscript.
3. A notion of *range* for refactorings is introduced that allows refactorings to be specified in a more localised fashion.
4. A small number of *patterns* for refactoring and techniques for proving them correct are identified.

CHAPTER 11: A FRAMEWORK FOR PROOF REFACTORING.

In this chapter, I present a concrete framework for proof refactoring that is *generic*, *extensible*, *formal*, and *declarative*. The framework is based on graph rewriting and bidirectional transformation and is implemented in a prototype tool called Polar. The work presented in this chapter was performed jointly with Dominik Dietrich. The original contributions of this chapter are:

1. The design and implementation of a framework for refactoring formal proof developments. Our prototype tool, Polar, currently supports two proof languages and over eight refactorings and is the first dedicated proof refactoring tool.
2. Furthermore, Polar is extensible in two directions: new proof languages and new refactorings can be added.

PART 3: CONCLUSIONS

CHAPTER 12: CONCLUSIONS AND FUTURE WORK

This chapter concludes the thesis by summarising the framework constructed in Part 1 and the approaches to refactoring described in Part 2 and draws conclusions before sketching some promising directions for future work.

1.3 HOW TO READ THIS THESIS

I have made a conscious effort to split this thesis into two parts, because I can imagine three types of readers:

1. My examiners and proof engineers with an interest in refactoring. I recommend reading from start to finish.
2. The proof language guru. For those interested in proof languages and their semantics, Part 1 will be of most interest to you and can be read from start to finish. Additionally, Chapter 6 can be read independently of Chapters 4 and 5.

3. The refactoring wizard. The read is slightly bumpier for those interested in refactoring. In general, Part 2 is for you; however, you may need to refer back to Part 1 to get a feel for the syntax and semantics of Hiscript for Chapter 9 and Chapter 10. I provide back references to the appropriate syntax and semantics at the start of these chapters.

Part I

PROOF LANGUAGE FRAMEWORK

2

PROOF LANGUAGES AND HIERARCHY

2.1 INTRODUCTION

In mathematics, proofs serve a dual role: to convey assurance that the formula under inspection is valid; and, to explain *why* it is true. Often, the understanding conveyed by a beautiful proof is just as important as the assurance that it provides. But, looking back, the history of mathematics is littered with flawed proofs: from the myriad of attempts at proving Euler's characteristic for polyhedrons:

$$V - E + F = 2$$

that are so well described by Lakatos to the near fatal hole in Wiles' famous proof of Fermat's Last Theorem (Lakatos et al., 1976; Wiles, 1995).

Theorem proving is the science and art of constructing formal proofs on a computer, using a formal logical language to state properties and a *proof language* to construct the proof. The subfield of Interactive Theorem Proving (ITP) focuses on using rich logics to prove complex properties with the assistance of a human guide.

In the popular LCF style of ITP — pioneered by Milner in the 70's — all proofs boil down to applications of a small set of primitive logical rules (Gordon et al., 1978). These rules form the constructors for a datatype *thm* and a small, easily understood kernel enforces that this is the only way a *thm* can be constructed. Proofs constructed by theorem provers come with a very strong guarantee of correctness indeed.

As Russell and Whitehead found out, constructing any non-trivial proof from primitive rules is a tedious business (Whitehead and Russell, 1912). The other innovation of LCF was to introduce a language for constructing proof procedures from the inference rules. These procedures are known as *tactics* and the language is *ML* (Milner et al., 1990).

While theorem provers solved the problem of assurance, proofs in these systems do not convey the understanding of an informal proof. The low-level proofs are too large and primitive to convey the general idea, while the *proof scripts* written in the tactic languages are too convoluted to understand without evaluating the proof again (Harrison, 1996).

In the past 15 years, proof languages with a more distinctly mathematical style, often known as declarative languages, have been developed on top of the traditional tactic layer of many systems. Declarative languages are gaining popularity; for example, seventeen of the twenty-four entries to Isabelle's Archive of Formal Proofs in 2012 used the declarative language Isar as the proof language of choice (Archive of Formal Proofs, 2012).

In another direction, there is also a growing interest in providing hierarchical representations for formal proofs. Informal mathematical proofs have always been described at different levels of detail: a high-level overview of the whole proof; glossing over simple parts: 'it is trivial to show...'; omitting repeated proof effort: 'the rest of the cases are similar...'; and, providing detailed proofs for the difficult or interesting parts of the proof. Hierarchical formal proofs can be used in the same way, with one crucial difference: detailed, low-level proofs are always available, so the level of detail of a particular part of the proof is not restricted to what the author of a proof chose. The reader can 'zoom in' arbitrarily to the level of most detail.

In comparison with programming languages, proof languages have not been studied in such great depth. In the forthcoming chapters, I build a formal framework in which to study proof languages, by providing a formal semantics to the proof languages. Furthermore, by building on a hierarchical proof representation I was also able to study the role of hierarchy in sophisticated proof languages; discovering, for example, that many constructs in these languages lend themselves naturally to hierarchy.

Moreover, it is not enough to study individual proofs as many theorem provers also have a language for constructing a *theory* — collections of theorems and related definitions — and a collection of theories, which I will call *proof documents*.

In the next section, I motivate and introduce the hierarchical proof representation that I use throughout this thesis. Then, in Section 2.3, I briefly sketch the current field of proof languages. Finally, in Section 2.4, I detail my approach to developing the *proof language framework* and provide a road map for the rest of Part 1.

2.2 HIERARCHICAL PROOF

Mathematical proofs naturally have structure. Mathematicians often discuss a proof in terms of its various 'parts' . In his famous article *How to Write a Proof*, Lamport emphasises the importance of a hierarchical presentation for proofs, providing an example proof of the irrationality of $\sqrt{2}$ at different levels of detail (Lamport, 1993). At the most abstract level, the proof is as follows:

Theorem 1 (Irrationality of $\sqrt{2}$)**.** *There does not exist* r *in* $\mathbb{Q}$ *such that* $r^2 = 2$

Proof.

1. Assume: $r \in \mathbb{Q}$ and $r^2 = 2$.
2. Choose $m, n \in \mathbb{Z}$ such that $gcd(m, n) = 1$ and $r = m/n$.
3. Then 2 divides m.
4. And 2 divides n.
5. Contradiction to $gcd(m, n) = 1$.

□

But, we could also 'zoom in' on details of step 3, for instance:

3a. $m^2/n^2 = r^2$, by squaring both sides.

3b. $m^2/n^2 = 2$ by assumption.

3c. $m^2 = 2n^2$ by multiplying both sides by n^2.

3d. Now m^2 is even.

3e. Thus m is even, since the square of an odd number is never even.

3f. Therefore $m = 2k$ for some k.

3g. Thus 2 divides m.

should further explanation be required.

Denney et al. (2006) introduced *hierarchical proofs* or *hiproofs* for short as a means to present the proof trees constructed by tactic-based theorem provers in a hierarchical way. Hiproofs are represented graphically and one can imagine an implementation that allows the user to unfold and fold the hierarchical components at will.

The hierarchy can also serve to make explicit the relationship between tactic calls and the proof tree constructed by these tactics, reflecting the fact that tactics are traditionally composed from simpler tactics. An abstract example of a hiproof is given in Figure 2.1, where a, b, and c are called *atomic tactics* i.e. black boxes.

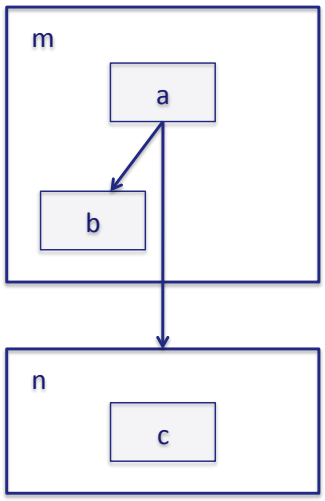

Figure 2.1: A Hiproof

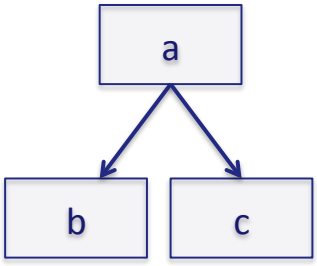

Figure 2.2: The skeleton

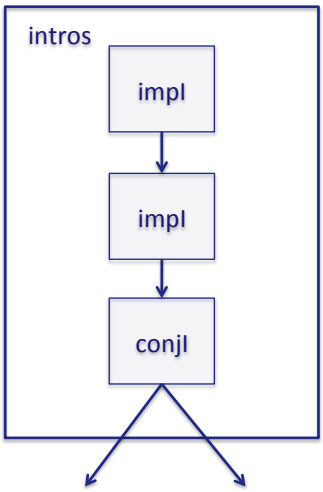

Figure 2.3: instance of *intros*

Figure 2.1 reads as follows: at the top, the abstract tactic *m* first applies an atomic tactic *a*. The tactic *a* produces two subgoals, the first of which is solved by the atomic tactic *b* within the application of *m*. Thus, the high-level view is that tactic *m* produces a single subgoal, which is then solved by the tactic *n*. The underlying proof tree, called the *skeleton*, is shown in Figure 2.2. Hiproofs can be given a denotational characterisation as a pair of forests, viewed as posets. One partial order provides a notion of hierarchy, the other of sequential composition. Conditions placed upon construction of hiproofs ensure that they can always be *unfolded* into the skeleton: a standard proof tree. These conditions, for example, require each hierarchical box to have a unique root node. As a more concrete example, Figure 2.3 shows the application of an *intros* tactic as a hiproof; the trailing edges are goals that must be

solved by composing other hiproofs to form a complete proof. For full details of this characterisation of hiproofs, I direct the reader to Denney et al. (2006).

In this thesis, however, I use an equivalent representation developed by Aspinall et al. (2010), that provides a linear term language equipped with an operational semantics. They also introduced a hierarchical tactic language called Hitac that I use for the tactic language in the Hiscript framework. So crucial are these formalisms to the rest of the thesis that Chapter 3 is fully devoted to them.

2.3 PROOF LANGUAGES

Freek Wiedijk's *Seventeen Provers of the World* provides an interesting comparison of the proof languages some of the most widely used theorem proving systems (Wiedijk, 2006). For each system, experienced practitioners submitted a proof of the irrationality of $\sqrt{2}$. I will use a few of the proofs as a means of introducing the broad styles of proof language and comparing them to the languages that I have developed.

2.3.1 *Procedural proofs*

The following proof is by John Harrison in his HOL Light system (Harrison, 1998).

```
let SQRT_2_IRRATIONAL = prove
 (`~rational(sqrt(&2))`,
  SIMP_TAC[rational; real_abs; SQRT_POS_LE; REAL_POS; NOT_EXISTS_THM] THEN
  REPEAT GEN_TAC THEN DISCH_THEN(CONJUNCTS_THEN2 ASSUME_TAC MP_TAC) THEN
  DISCH_THEN(MP_TAC o AP_TERM `\x. x pow 2`) THEN
  ASM_SIMP_TAC[SQRT_POW_2; REAL_POS; REAL_POW_DIV; REAL_POW_2; REAL_LT_SQUARE;
               REAL_OF_NUM_EQ; REAL_EQ_RDIV_EQ] THEN
  ASM_MESON_TAC[NSQRT_2; REAL_OF_NUM_EQ; REAL_OF_NUM_MUL]);;
```

This typical LCF-style *procedural* proof is dense and difficult to understand without proper training. This *implicit* style of proof, consisting of instructions to the system to find the proof, is a function of type *tactic* in OCaml, the implementation language of the system. Since the tactic language is a full-blown programming language, the user of the system has a lot of flexibility to write sophisticated tactics, such as decision procedures. This is one of the reasons that the HOL systems are so powerful. At the same time, this power makes it difficult to reason about the tactics and the style of proof makes it difficult to understand the proof. Furthermore, these procedural proofs are often brittle: slight changes to definitions, formulae or tactics can trigger unpredictable changes in the proofs and it becomes difficult to trace the source of divergence from the original proof and to make an appropriate patch. The individual tactics like SIMP_TAC and GEN_TAC are composed by functions called *tacticals*, such as REPEAT and THEN.

The Hitac language follows this style of proof. It differs from the traditional HOL languages, though, as Hitac is a Domain Specific Language with a clean semantics that explicitly constructs hiproofs. On the other hand, Hitac lacks the power of a full-blown functional programming language.

2.3.2 *Declarative languages*

The following is a proof in Makarius Wenzel's Isar language for the Isabelle system (Wenzel, 1999; Nipkow et al., 2002).

```
theorem "p ∈ prime ⟹ sqrt (real p) ∉ ℚ"
proof
  assume p_prime: "p ∈ prime"
  then have p: "1 < p" by (simp add: prime_def)
  assume "sqrt (real p) ∈ ℚ"
  then obtain m n where
      n: "n ≠ 0" and sqrt_rat: "|sqrt (real p)| = real m / real n"
    and gcd: "gcd (m, n) = 1" ..
  from n and sqrt_rat have "real m = |sqrt (real p)| * real n" by simp
  then have "real (m²) = (sqrt (real p))² * real (n²)"
    by (auto simp add: power_two real_power_two)
  also have "(sqrt (real p))² = real p" by simp
  also have "... * real (n²) = real (p * n²)" by simp
  finally have eq: "m² = p * n²" ..
  then have "p dvd m²" ..
  with p_prime have dvd_m: "p dvd m" by (rule prime_dvd_power_two)
  then obtain k where "m = p * k" ..
  with eq have "p * n² = p² * k²" by (auto simp add: power_two mult_ac)
  with p have "n² = p * k²" by (simp add: power_two)
  then have "p dvd n²" ..
  with p_prime have "p dvd n" by (rule prime_dvd_power_two)
  with dvd_m have "p dvd gcd (m, n)" by (rule gcd_greatest)
  with gcd have "p dvd 1" by simp
  then have "p ⩽ 1" by (simp add: dvd_imp_le)
  with p show False by simp
qed
```

This *declarative* proof could be read and understood by a mathematician. In the proof, goals and assumptions are explicitly stated, and the proof is mostly in a forwards style. The explicit justifications that are provided to the prover are given with the **by** command. These justifications can be seen as *tactics* in Isabelle's underlying tactic language. Isar is now firmly established as the proof language of choice in the Isabelle system. For example, seventeen of the twenty-four entries submitted to the Archive of Formal Proof in 2012 used Isar as the predominant proof language[1].

The history of declarative proof languages dates from the Mizar system, which was developed around the same time as LCF and the HOL family, but with a completely different language for constructing formal proofs (Rudnicki, 1992). The Mizar proof language more closely resembles traditional mathematical proofs, where assumptions and facts are stated in a declarative way. The Mizar system can then be executed and would attempt to fill in any gaps left in the script using a fixed proof procedure, or it would fail.

Many other proof assistants, such as Coq (Corbineau, 2008), HOL Light (Wiedijk, 2012), and Ωmega, (Dietrich, 2011) have spawned declarative languages. As another

1 This is a difficult comparison to make, since Isar allows procedural proof to be mixed in with declarative proofs. In practice, most submissions used a mix of the two styles.

approach to making proofs readable, Matita provides a natural language rendering of proof objects (Coen, 2010).

The Hiscript declarative language in Chapter 4 is similar to Isar, taking inspiration from it for syntax and language constructs; although, since Hiscript is intended for formal study of proof language semantics, it lacks some of the important abbreviations and language constructs for real-life use.

2.3.3 *SSReflect*

Finally, a variant on both styles is George Gonthier's SSReflect language for the Coq proof assistant (Gonthier et al., 2008; The Coq development team, 2004). The language was developed for his proof of the Four Colour Theorem and has since been refined into a mature proof language. An SSReflect proof of the irrationality of $\sqrt{2}$ is shown below:

```
Theorem sqrt2_irrational : ~(EX f : frac | `f = sqrt 2`).
Proof.
Move=> [f Df]; Step [Hf22 H2f2]: `(mulf f f) = F2`.
  Apply: (eqr_trans (fracr_mul ? ? ?)); Apply: eqr_trans (fracrz R (Znat 2)).
  By Apply: eqr_trans (square_sqrt (ltrW (ltr02 R))); Apply mulr_morphism.
Step Df2: (eqf f2 (mulf f f)) By Apply/andP; Split; Apply/(fracr_leqPx R ? ?).
Move: f Df2 {Hf22 H2f2 Df} => [d m]; Rewrite: /eqf /= -eqz_leq; Move/eqP.
Rewrite: scalez_mul -scalez_scale scalez_mul mulzC {-1 Zpos}lock /= -lock.
Step []: (Zpos (S d)) = (scalez d (Znat 1)).
  By Apply esym; Apply: eqP; Rewrite scalez_pos; Elim d.
Step [n []]: (EX n | (mulz (Zpos n) (Zpos n)) = (mulz m m)).
  Case: m => [n | n]; LeftBy Exists n.
  By Exists (S n); Rewrite: -{1 (Zneg n)}oppz_opp mulz_oppl -mulz_oppr.
Pose i := (addn (S d) n); Move: (leqnn i) {m}; Rewrite: {1}/i.
Elim: i n d => // [i Hrec] n d Hi Dn2; Move/esym: Dn2 Hi.
Rewrite: -{n}odd_double_half double_addnn !zpos_addn; Move/half: n (odd n) =>
    n.
Case; [Move/((congr oddz) ? ?) | Move/((congr halfz) ? ?)].
  By Rewrite: !mulz_addr oddz_add mulzC !mulz_addr oddz_add !oddz_double.
Rewrite: add0n addnC -addnA add0z mulz_addr !halfz_double mulzC mulz_addr.
Case: n => [|n] Dn2 Hi; LeftBy Rewrite: !mulz_nat in Dn2.
Apply: Hrec Dn2; Apply: (leq_trans 3!i) Hi; Apply: leq_addl.
Qed.
```

On first appearance, this proof is as unreadable as the HOL Light version; however, the SSReflect style of proof attempts to ensure each line in the proof, terminated by a full-stop, takes a clear mathematical step. In this way, proofs in SSReflect aim to be readable upon replay and many of the constructs in the language are designed to help write proofs in this style.

I describe this language in more detail and provide a formal semantics to a subset of SSReflect using Hitac in Chapter 6.

2.4 SUMMARY

This part presents both an investigation into both proof language semantics and the role of hierarchy in proof. Furthermore, it consists of two parallel strands of language development on top of the procedural Hitac language:

1. The Hiscript framework, which consists of a declarative proof language, described in Chapter 4, and a theory language, described in Chapter 5.
2. The eSSence proof language which implements a subset of the SSReflect language for Coq, which is described in Chapter 6.

I provide a background primer on hiproofs and describe the tactic language Hitac in the next chapter.

3

HIPROOFS AND HITACS

3.1 INTRODUCTION

Aspinall et al. (2010) have laid the foundations for the framework for formal proof development: providing a convenient linear representation for the Hiproof formalism — as a term language, inspired by the underlying categorical model of a proof — and a 'traditional' LCF-style tactic language — called Hitac — that can be executed (or evaluated) on goals to construct (hi)proofs[1]. This chapter reproduces the relevant material from Aspinall et al. (2010) and introduces some additional language constructs and concepts that are used in the rest of this thesis

CHAPTER MAP The next section provides a detailed presentation of hiproofs. I then introduce the Hitac language and its formal semantics in Section 3.3 and summarise in Section 3.4.

CONTRIBUTIONS

1. The term languages for hiproofs and hitacs are extended to include a *swap* operator that inverts the order in which goals can be proved.
2. The hitac language is further extended with a 'proof by lemma' construct.
3. A notion of normal form for hiproofs is provided.
4. A notion of *well-formedness checking* for hitacs is introduced to detect some simple causes of failure in a static way; that is, without evaluating the hitac against a goal. It can be seen as very basic type-checking.

3.2 HIERARCHICAL PROOF

In this section, I recap the presentation of hiproofs given by Aspinall et al. (2010), extending it with one new construct. I also provide several example hiproofs and introduce a normal form for hiproofs.

1 I will often call a term in the hitac language a *hitac tactic* or even just a *hitac*.

3.2.1 *A hiproof term language*

Hiproofs are intended to be a generic representation for (hierarchical) proof trees. They do not commit to a specific logical system. Rather, they use a *derivation system*, which simply gives an abstract notion of *goals* and *atomic tactics*.

Definition 1 (Derivation systems and atomic goals)**.** A *derivation system*, $\mathcal{D}$, is a pair of sets of *goals* $\mathcal{G}$ and *atomic tactics*, $\mathcal{A}$. Atomic tactics simply map a list of premises to a conclusion and can be viewed as inference rule schemas:

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

stating that the atomic tactic a, given proofs of the *subgoals* $\gamma_1, \ldots, \gamma_n$, produces a proof of the goal γ. With a derivation system, hiproofs also assume a matching relation, written $\sim$, that allows us to compare goals.

Convention. Metavariables γ, γ_1, $\ldots$, γ_n will range over goals. Lists of goals will often be written as g, g_1, $\ldots$, g_n. Similarly, atomic tactics will be written as a, a_1, $\ldots$, a_n.

The relationship between rule schemas and their instances is not prescribed; however, each instance must have the same *arity*. That is, they must have the same number of premises. It is often useful, particularly in the setting of interactive proof, to view atomic tactics backwards: taking an input goal and producing a list of *subgoals*.

Figure 3.1 introduces the term language for hiproofs. This term language gives a compact, linear representation for the underlying graphical structures.

Hiproof syntax

s	$::=$	a	an atomic tactic
	$\mid$	id	identity tactic
	$\mid$	$[l]\ s$	labelled hiproof
	$\mid$	$s\ ;\ s$	sequential composition of proofs
	$\mid$	$s \otimes s$	parallel composition of proofs: 'tensor'
	$\mid$	$\langle\rangle$	empty
	$\mid$	$swap$	swap the order of two subgoals

Figure 3.1: Linear syntax for hierarchical proofs

Atomic tactics, $a \in \mathcal{A}$ are the core elements of the derivation system. There is a special identity hiproof *id*, which maps a goal to itself and the empty hiproof is represented by $\langle\rangle$. Hiproofs can be composed in two ways: *sequentially* using the sequencing operator (;) or in parallel using the tensor operator ($\otimes$). Graphically, this corresponds to composing hiproofs using arrows or placing hiproofs side-by-side. Finally, one can enclose a hiproof in a labelled box using the labelling operator

$[l]\ s$. Graphically, this corresponds to the hierarchical nesting of hiproofs. The labels, $l \in \mathcal{L}$ are taken from an unspecified set of identifiers. Labelling is not a binder: in the hiproof $[l]\ s$ the label l may be reused. A hiproof term without a subterm of this form is called *label-free*. To reduce bracketing, each hiproof combinator has a binding power. Labelling binds weakest, followed by sequencing, with tensoring binding most tightly. Finally, I introduce a new construct, *swap*, to invert the order of two goals. This construct allows the order in which the goals were solved to be represented explicitly.

To illustrate this term language, I provide a hiproof term for two simple hiproofs. First, however, I provide a derivation system, which for simplicity is kept abstract and consists of the following three atomic tactics:

A simple set of atomic tactics

$$\frac{\gamma_2 \quad \gamma_3}{\gamma_1}\ a \qquad \frac{}{\gamma_2}\ b \qquad \frac{}{\gamma_3}\ c$$

Figure 3.2 shows a possible hiproof of the goal γ_1. It can be read as 'the abstract tactic m first applies an atomic tactic a'. The tactic a produces two subgoals, the first of which is solved by the atomic tactic b within the application of m. Thus, the high-level view is tactic m produces a single subgoal, which is then solved by the tactic n. Using the term language, the hiproof (in Figure 3.2) can be represented as:

$$([m]\ a\ ;\ b \otimes id)\ ;\ [n]\ c \tag{3.1}$$

Figure 3.3 shows a different possible hiproof that solves the goal γ_1, where the labelled box n is now nested. This would be represented in the syntax as:

$$[m]\ a\ ;\ b \otimes ([n]\ c) \tag{3.2}$$

To illustrate the *swap* operation, consider Figure 3.4, which is the same hiproof as Figure 3.3 except that the subgoals generated by a are proved in a different order[2]. This would be represented in the syntax as:

$$[m]\ a\ ;\ swap\ ;\ ([n]\ c) \otimes b \tag{3.3}$$

3.2.2 *Hiproof validation*

Since hiproofs decorate normal proof trees with hierarchical boxes, they require some guarantee that, underneath, it is still a proof. This notion is called *validation* (of a hiproof). Hiproof validation is defined on finite lists $g \in \mathcal{G}^*$ of goals $[\gamma_1, \ldots, \gamma_n]$ where each $\gamma_i \in \mathcal{G}$ and the concatenation of two lists is written as $g_1 @ g_2$ and the empty list as $[]$. The length of a goal list is its *arity* and is written $g : n$. Thus, a

2 From now on, I will omit the explicit goal labelling in hiproofs.

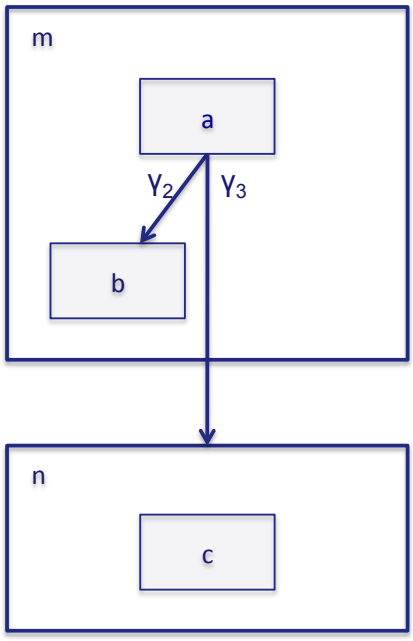

Figure 3.2: The canonical hiproof example

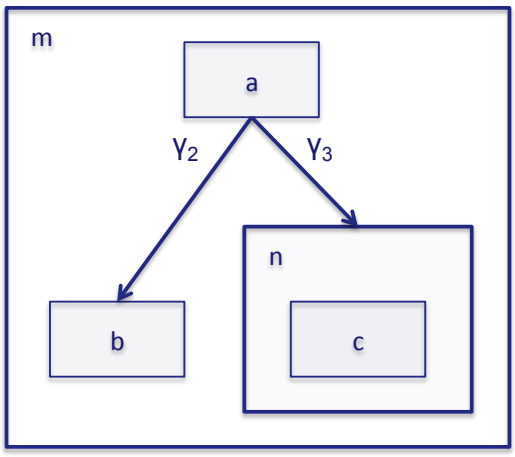

Figure 3.3: Nested hierarchy example

hiproof s is a *valid* proof of g_1 with remaining subgoals g_2 if $s \vdash g_1 \longrightarrow g_2$. The inductive definition for hiproof validation is given by the rules in Figure 3.5.

The notion of validation not only checks that the underlying proof is indeed a proof, but also ensures that the hierarchical structure is well-formed with respect to the characterisation provided by Denney et al. (2006), as mentioned in Section 2.2. Finally, it makes it clear what it means to be a partial hiproof: that is, a proof that leaves some goals remaining unsolved.

The rule V-Lab states that each hierarchical box must have a single incoming goal, as expected. The swap operation built into hiproofs is the simplest possible: it simply rotates a list of two goals. To perform more complicated reorderings, one must compose it. For example $id \otimes swap$ will swap the second and third goal in a three-goal list and $swap \otimes id \; ; \; id \otimes swap$ rotates a list of three goals to the left by one goal. It is interesting to note that, since a hiproof consists of schematic rules, the same hiproof could be valid for many different input goals. A hiproof that validates a non-empty set of subgoals is called a *partial hiproof*[3] and I will often make no distinction. Thus the example hiproof in Figure 3.3 is valid:

$$([m]\ a \; ; \; b \otimes id) \; ; \; [n]\ c \vdash [\gamma_1] \longrightarrow [].$$

Additionally, the partial hiproof $[m]\ a \; ; \; b \otimes id$ is valid:

$$[m]\ a \; ; \; b \otimes id \vdash [\gamma_1] \longrightarrow [\gamma_3].$$

We can also scrap all the hierarchy to get an old-fashioned proof. The underlying proof tree is called the *skeleton*. The resulting (label-free) hiproof is still valid. Interestingly, both the hiproofs in Figures 3.2 and 3.3 have the same skeleton, visualised in Figure 3.6. The skeleton for Figure 3.4 is identical except the swapping of the atomic tactics b and c is still present.

3 Informally, it has *danglers*.

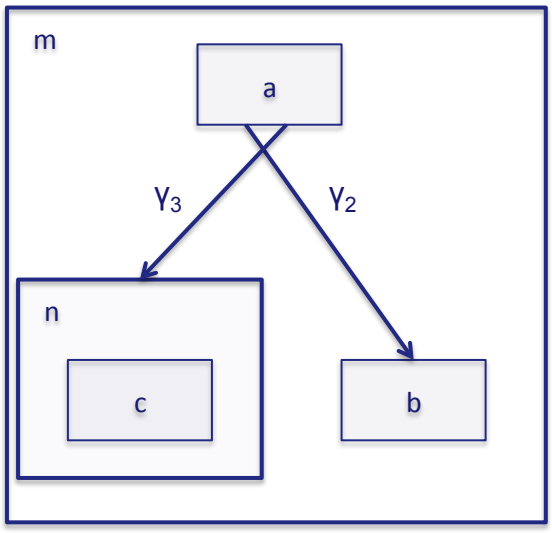

Figure 3.4: Swapping the order of subgoals

3.2.3 *A hiproof normal form*

In this section, I state a possible normal form for hiproofs and a set of rewrite rules that terminate with that form. In this section, the hiproofs are assumed to be *swap-free*. I have already shown two ways in which a hiproof of the goal γ_1 can be constructed (hiproofs 3.1 and 3.2). Consider, however, what the skeleton of each would look like:

$$skeleton(([m]\ a\ ;\ b\ \otimes\ id)\ ;\ [n]\ c) \quad = \quad a\ ;\ (b\ \otimes\ id)\ ;\ c \tag{3.4}$$

$$skeleton([m]\ a\ ;\ b\ \otimes\ [n]\ c) \quad = \quad a\ ;\ b\ \otimes\ c \tag{3.5}$$

Thus, the term language can represent label-free hiproofs in multiple ways. Clearly $a\ ;\ b\ \otimes\ c$ is a more concise hiproof. The identity is simply a 'legacy' of the *wiring* of the two hierarchical boxes m and n. There is further *ambiguity* in label-free hiproof representations. Consider the derivation system extended with the following atomic tactics:

$$\frac{\gamma_2}{\gamma_1}\ d \qquad\qquad \overline{\gamma_2}\ e$$

Then, the hiproof in Figure 3.7 solves a pair $[\gamma_1, \gamma_1]$ of identical goals γ_1. This hiproof can be represented as:

$$(d\ \otimes\ d)\ ;\ (e\ \otimes\ e) \tag{3.6}$$

or as follows:

$$(d\ ;\ e)\ \otimes\ (d\ ;\ e) \tag{3.7}$$

$$\frac{\frac{\gamma_1 \ldots \gamma_n}{\gamma}\ a \in \mathcal{A}}{a \vdash \gamma \longrightarrow [\gamma_1, \ldots, \gamma_n]} \quad \text{(V-Atomic)}$$

$$id \vdash [\gamma] \longrightarrow [\gamma] \quad \text{(V-Id)}$$

$$\frac{s \vdash [\gamma] \longrightarrow g}{[l]\ s \vdash [\gamma] \longrightarrow g} \quad \text{(V-Lab)}$$

$$\frac{s_1 \vdash g_1 \longrightarrow g \qquad s_2 \vdash g \longrightarrow g_2}{s_1\ ;\ s_2 \vdash g_1 \longrightarrow g_2} \quad \text{(V-Seq)}$$

$$\frac{s_1 \vdash g_1 \longrightarrow g_1' \qquad s_2 \vdash g_2 \longrightarrow g_2'}{s_1 \otimes s_2 \vdash g_1 \mathbin{@} g_2 \longrightarrow g_1' \mathbin{@} g_2'} \quad \text{(V-Tens)}$$

$$\langle\rangle \vdash [] \longrightarrow [] \quad \text{(V-Empty)}$$

$$swap \vdash [\gamma_1, \gamma_2] \longrightarrow [\gamma_2, \gamma_1] \quad \text{(V-Swap)}$$

Figure 3.5: Hiproof validation relation

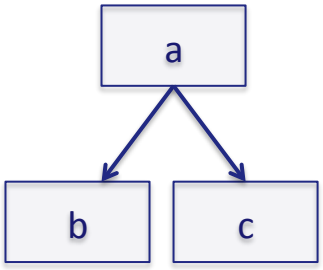

Figure 3.6: The skeleton of the example hiproofs

or even as follows:

$$(d \otimes id)\ ;\ (e \otimes d)\ ;\ e$$

It is not obvious which of the first two representations is preferable. The first possibility can be read as 'take one step for each subgoal sequentially' and the second as 'solve one subgoal before moving on to the rest'. This latter understanding is closer to what often happens in practice, and it is the representation I take as preferable[4].

Thus, consider the following 'distributivity' rule:

$$(s_1 \otimes s_2)\ ;\ (s_3 \otimes s_4) = (s_1\ ;\ s_3) \otimes (s_2\ ;\ s_4) \tag{3.8}$$

to aid the normalisation process. Application of this rule (left to right) on hiproof 3.6 — where s_1 and s_2 are the atomic tactic d and s_3 and s_4 are the atomic tactic e — will obtain the desired form, hiproof 3.7.

4 On a practical note, it can also be thought of as allowing the most parallelisation as the tensor operation is always the root of the term tree and each subgoal can be forked to different proof processes.

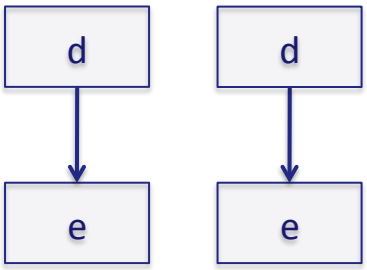

Figure 3.7: A hiproof with multiple term representations

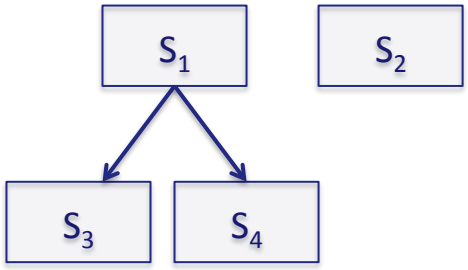

Figure 3.8: A hiproof where the general distributivity rule fails

Life is not so simple as could be wished, however. Consider the hiproof in Figure 3.8. This can be represented as $(s_1 \otimes s_2)\ ;\ (s_3 \otimes s_4)$ — making it a candidate for transformation — but the result, $(s_1\ ;\ s_3) \otimes (s_2\ ;\ s_4)$, would not be a valid, let alone equivalent, hiproof. The reason being that the *arity* of atomic tactics (and therefore the composite hiproofs) is the critical factor in determining the structure and the syntax ignores this completely, though validation does not. The next section captures this information with an *annotated* hiproof.

3.2.3.1 *Annotated hiproofs*

To recover this lost information, I define a notion of *annotation* for a hiproof, writing

$$s : n \Rightarrow m$$

to say a hiproof s has arities $n \Rightarrow m$, where $n, m \in \mathbb{N}$ if $s \vdash g_1 \longrightarrow g_2$ and $g_1 : n$ and $g_2 : m$. That is, it counts the number of goals operated on by the hiproof. Figure 3.9 gives an inductive definition of the annotation relation.

These rules are straightforward: tensoring sums the inputs and outputs, sequencing passes the outputs of the first as inputs to the second, labelling requires an input of one goal, and atomic tactics take a single input and the number of subgoals as output. The identity hiproof always takes a single input and provides a single output; and, the empty hiproof works on empty goal lists so as annotation $0 \Rightarrow 0$.

To illustrate the annotation further, I provide arities for some simple hiproofs:

$$\frac{\frac{\gamma_1 \cdots \gamma_n}{\gamma}\ a \in \mathcal{A}}{a : 1 \Rightarrow n} \quad \text{(T-Atomic)}$$

$$id : 1 \Rightarrow 1 \quad \text{(T-Identity)}$$

$$\frac{s : 1 \Rightarrow o}{[l]\ s : 1 \Rightarrow o} \quad \text{(T-Lab)}$$

$$\frac{s_1 : i \Rightarrow j \qquad s_2 : j \Rightarrow k}{s_1\ ;\ s_2 : i \Rightarrow k} \quad \text{(T-Seq)}$$

$$\frac{s_1 : i_1 \Rightarrow j_1 \qquad s_2 : i_2 \Rightarrow j_2}{s_1\ \otimes\ s_2 : i_1 + i_2 \Rightarrow j_1 + j_2} \quad \text{(T-Tens)}$$

$$\langle\rangle : 0 \Rightarrow 0 \quad \text{(T-Empty)}$$

$$swap : 2 \Rightarrow 2 \quad \text{(T-Swap)}$$

Figure 3.9: Hiproof annotation rules

$$\begin{aligned} a &: 1 \Rightarrow 2 \\ a\ ;\ b\ \otimes\ id &: 1 \Rightarrow 1 \\ [m]\ a\ ;\ b\ \otimes\ id &: 1 \Rightarrow 1 \\ d\ \otimes\ d &: 2 \Rightarrow 2 \\ (d\ \otimes\ d)\ ;\ (e\ \otimes\ e) &: 2 \Rightarrow 0 \\ a\ ;\ a\ \otimes\ a &: 1 \Rightarrow 4 \end{aligned}$$

All these hiproofs except the final hiproof are also valid. Validation implies they can be annotated, but not the converse. The annotation simply ensures that the boxes could fit together. The following hiproofs cannot be annotated:

$$a\ ;\ b$$

$$[m]\ d\ \otimes\ d\ ;\ e\ \otimes\ e$$

since sequencing must match output goals to input goals and labelling requires a single input goal, not two.

3.2.3.2 *A Hiproof normal form*

The following definition captures the structure of the normal form hinted at in the previous sections, where:

1. All unnecessary empty hiproofs are removed.
2. All unnecessary identities are removed.

3. The hiproofs are maximally parallel: that is, tensors are distributed upwards in the term tree as much as possible. For this reason I call the normal form *Tensor Normal Form*.

Definition 2 (Tensor Normal Form)**.** A hiproof s is in Tensor Normal Form (TNF) — writing $\equiv$ for syntactic equality — if:

1. $s \equiv \langle\rangle$
2. if $s : 1 \Rightarrow n$, then one of the following holds:
   a) $s \equiv id$;
   b) $s \equiv a$, for some $a \in \mathcal{A}$;
   c) $s \equiv [l]\ s'$ and s' is in TNF;
   d) $s \equiv a\ ;\ s'$ where $a \in \mathcal{A}$ and s' is in TNF;
   e) $s \equiv ([l]\ s')\ ;\ s''$ and s' and s'' are in TNF;
3. if $s : n \Rightarrow m$, where $s \vdash [\gamma_1, \ldots, \gamma_n] \longrightarrow g$ and $n > 1$ then:

   $$s \equiv s_1 \otimes \ldots \otimes s_n$$

   such that $\forall\ i\ .\ s_i \vdash \gamma_i \longrightarrow g_i$ and $g_1 @ \ldots @ g_n = g$ and each s_i is in TNF.

The definition identifies three cases of hiproof type: zero input, single input, and multiple input and ensures they fit the right pattern. According to the definition, then, each of the hiproofs below are in normal form:

$$([m]\ a\ ;\ b\ \otimes\ id)\ ;\ [n]\ c$$

$$a\ ;\ b\ \otimes\ c$$

$$(d\ ;\ e)\ \otimes\ (d\ ;\ e)$$

$$(a\ ;\ b\ \otimes\ c)\ \otimes\ (d\ ;\ e) \tag{3.9}$$

The following hiproofs are not in normal form:

$$id\ ;\ a$$

$$a\ \otimes\ d\ ;\ b\ \otimes\ c\ \otimes\ e$$

The first example has type $1 \Rightarrow 2$, but doesn't match any pattern in case 2. The second example needs to have the tensors distributed over the sequential composition operator. Note that hiproof 3.9 (visualised in Figure 3.10) is the normal form for this example. As a final example, the following hiproof is not in TNF because the (sub) hiproof inside the labelled box m is not in TNF:

$$[m]\ d\ ;\ id\ ;\ e.$$

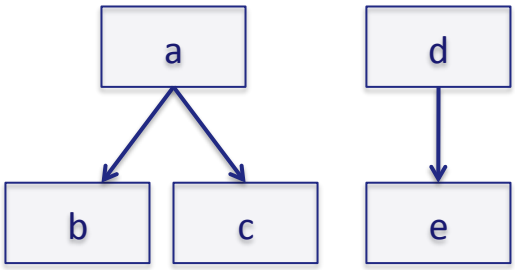

Figure 3.10: Another hiproof with many representations

3.2.3.3 *Transformation rules*

I now provide a set of transformation rules that terminate with a hiproof in TNF. I give the core set of rewrite rules in Figure 3.11, which is extended with congruence rules in Figure 3.12. The rewrite rules are applied left to right and conditional rewrites are represented as inference rules (the rule under the line can be applied if the conditions above the line are satisfied).

$$id \;;\; s \xrightarrow{tnf} s \qquad \text{(TNF-ID-L)}$$

$$s \;;\; id \xrightarrow{tnf} s \qquad \text{(TNF-ID-R)}$$

$$\langle\rangle \otimes s \xrightarrow{tnf} s \qquad \text{(TNF-EMP-L)}$$

$$s \otimes \langle\rangle \xrightarrow{tnf} s \qquad \text{(TNF-EMP-R)}$$

$$\frac{s : n \Rightarrow 0}{s \;;\; \langle\rangle \xrightarrow{tnf} s} \qquad \text{(TNF-EMP)}$$

$$\frac{s_1 : n \Rightarrow n' \qquad s_2 : n' \Rightarrow m}{(s_1 \otimes s_1') \;;\; (s_2 \otimes s_2') \xrightarrow{tnf} (s_1 \;;\; s_2) \otimes (s_1' \;;\; s_2')} \qquad \text{(TNF-1)}$$

$$\frac{s_2 : n \Rightarrow 0}{(s_1 \otimes s_2) \;;\; s_3 \xrightarrow{tnf} (s_1 \;;\; s_3) \otimes s_2} \qquad \text{(TNF-2-L)}$$

$$\frac{s_1 : n \Rightarrow 0}{(s_1 \otimes s_2) \;;\; s_3 \xrightarrow{tnf} s_1 \otimes (s_2 \;;\; s_3)} \qquad \text{(TNF-2-R)}$$

Figure 3.11: Hiproof transformation rules

The first five rules (TNF-ID-L to TNF-EMP) are straightforward to understand. The rule TNF-1 is exactly what is needed to rewrite hiproof 3.6 to its normal form. The condition ensures that all of the outputs of s_1 get passed onto s_2, which is crucial to ensuring equivalence of the rewritten hiproof. The rules TNF-2-L and TNF-2-R are new. They say that if one side of a tensor product solves all its inputs (has an arity of

$$\frac{s \xrightarrow{tnf} s'}{[l]\ s \xrightarrow{tnf} [l]\ s'}$$

$$\frac{s_1 \xrightarrow{tnf} s_1'}{s_1 \otimes s_2 \xrightarrow{tnf} s_1' \otimes s_2}$$

$$\frac{s_2 \xrightarrow{tnf} s_2'}{s_1 \otimes s_2 \xrightarrow{tnf} s_1 \otimes s_2'}$$

$$\frac{s_1 \xrightarrow{tnf} s_1'}{s_1\ ;\ s_2 \xrightarrow{tnf} s_1'\ ;\ s_2}$$

$$\frac{s_2 \xrightarrow{tnf} s_2'}{s_1\ ;\ s_2 \xrightarrow{tnf} s_1\ ;\ s_2'}$$

Figure 3.12: Hiproof transformation congruence rules

0), then it can safely move outside a sequencing operator. To see why these two rules are needed, consider the skeleton of hiproof 3.1:

$$a\ ;\ b \otimes id\ ;\ c$$

This hiproof is not yet in TNF as one can replace the identity with the atomic tactic c that it maps its goal to. In Definition 2, this hiproof matches case 2d), but s' (in this case the subterm $b \otimes id\ ;\ c$) is not in TNF. However, the distributivity rule TNF-1 requires a tensor on each side of the sequential composition operator, which is not present. The congruence rules allow rewriting inside subproofs. Since both tensor and sequencing operators are associative, rewriting is allowed to choose the appropriate bracketing. In the literature, this is typically performed by forming a list representation for associative binary operators, then rewriting sublists.

These rules are illustrated with a simple (somewhat contrived) example, which is shown in Figure 3.13 with explicit identities:

$$(a \otimes id)\ ;\ (b \otimes id \otimes a)\ ;\ (c \otimes b \otimes c) \tag{3.10}$$

To normalise this, the rewrite rules are applied exhaustively. Here is one possible trace (ignoring the congruence rule applications and associativity):

1. Let $s_1 \equiv a$, $s_1' \equiv id$, $s_2 \equiv b \otimes id$, and $s_2' \equiv a$, then by TNF-1 :

$$(a \otimes id)\ ;\ (b \otimes id \otimes a)\ ;\ (c \otimes b \otimes c) \xrightarrow{tnf} (a\ ;\ b \otimes id) \otimes (id\ ;\ a)\ ;\ (c \otimes b \otimes c)$$

2. A second rewrite is to replace $id\ ;\ a$ using TNF-ID-L:

$$(a\ ;\ b \otimes id) \otimes (id\ ;\ a)\ ;\ (c \otimes b \otimes c) \xrightarrow{tnf} (a\ ;\ b \otimes id) \otimes a\ ;\ (c \otimes b \otimes c)$$

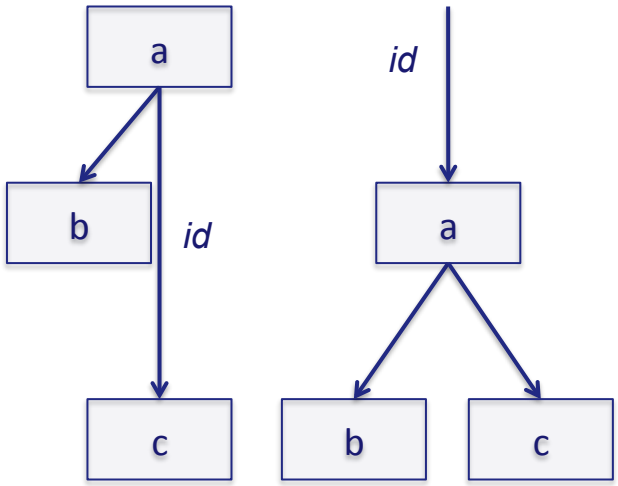

Figure 3.13: Rewriting example, before any rules are applied

3. Next, applying TNF-1 again with $s_1 \equiv a \ ; \ b \otimes id$, $s_1' \equiv a$, $s_2 \equiv c$, and $s_2' \equiv b \otimes c$ obtains:

$$(a \ ; \ b \otimes id) \otimes a \ ; \ (c \otimes b \otimes c) \xrightarrow{tnf} (a \ ; \ b \otimes id \ ; \ c) \otimes (a \ ; \ b \otimes c)$$

4. Now, the only rewrite rule that can be applied is TNF-2-R where $s_1 \equiv b$, $s_2 \equiv id$, and $s_2 \equiv c$ to obtain:

$$(a \ ; \ b \otimes id \ ; \ c) \otimes (a \ ; \ b \otimes c) \xrightarrow{tnf} (a \ ; \ b \otimes (id \ ; \ c)) \otimes (a \ ; \ b \otimes c)$$

5. Finally, TNF-ID-L can be used again to remove the identity in $id \ ; \ c$:

$$(a \ ; \ b \otimes (id \ ; \ c)) \otimes (a \ ; \ b \otimes c) \xrightarrow{tnf} (a \ ; \ b \otimes c) \otimes (a \ ; \ b \otimes c)$$

6. At this point, the hiproof (and its validation) is:

$$(a \ ; \ b \otimes c) \otimes (a \ ; \ b \otimes c) \vdash [\gamma_1, \gamma_1] \longrightarrow []$$

which cannot be rewritten any further. It is in TNF: each side of the main tensor $(a \ ; \ b \otimes c)$ validates a single goal (γ_1) and they are themselves in TNF.

3.2.3.4 *Properties of the transformation rules*

First:

Theorem 2 (Correctness of rewrite rules). *If* $s \vdash g \longrightarrow g'$ *and* $s \xrightarrow{tnf} s'$ *then* $s' \vdash g \longrightarrow g'$.

PROOF. This follows by a simple case-analysis on the rules.

Consider the rewrite system induced by this set of rules, which is written $s \xrightarrow{tnf}{}^{*} s'$, then the following properties hold:

Theorem 3 (Termination). *The rewrite system generated from the rules in Figure 3.11 and 3.12, applied transitively and associatively is terminating.*

Proof. Termination of the rules is proved by a simple measure function $\phi(s)$, which always decreases with applications of the rules. In particular, ϕ is defined as follows (in english): the number of symbols in each sequence multiplied by two plus the number of symbols in each tensor product. For example, in the rule:

$$s \ ; \ id \xrightarrow{tnf} s$$

the number of symbols in the sequence is 2; there are no tensor symbols; therefore $\phi(s) = 2 * 2 = 4$ on the LHS. It is zero on the RHS.

For the more complicated rule:

$$\frac{s_1 : \ n \Rightarrow 0}{(s_1 \ \otimes \ s_2) \ ; \ s_3 \xrightarrow{tnf} s_1 \ \otimes \ (s_2 \ ; \ s_3)}$$

We have (on LHS):

- Number sequenced symbols is 4 (s_1, $\otimes$, s_2, *and* s_3);
- Number tensored symbols is 2 (s_1, s_2).

This gives $\phi(s) = 4 * 2 + 2 = 10$. On the RHS, we have $\phi = 8$. □

Theorem 4 (Confluence)**.** *The set of rewrite rules form a confluent rewrite system.*

Proof. Firstly, since the rewrite rules are terminating, by Newman's lemma it is sufficient to show weak confluence (Newman, 1942, Theorem 2). The standard technique is *critical pair analysis* and I follow this. For reference:

Definition 3 (Critical Pairs)**.** Let $x \to y$ and $u \to v$ be two rules of a term rewriting system, and suppose these rules have no variables in common. If they do, rename the variables. If x_1 is a subterm of x (or the term x itself) such that it is not a variable, and the pair (x_1, u) is unifiable with the most general unifier θ, then $y\theta$ and the result of replacing $x_1\theta$ in $x\theta$ by $v\theta$ are called a critical pair.

Then, if all critical pairs are joinable — that is, they can be rewritten to the same term — the system is confluent. Reasoning about the critical pairs also needs preconditions for the rewrites to be taken into account. Here, only one of the critical pairs arising between the rules TNF-1 and TNF-EMP-L is considered. The rest are similar. Let the rule $x \to y$ be TNF-1 and $u \to v$ be TNF-EMP-L then the subterm $s_1 \ \otimes \ s_1'$ has a *mgu* with TNF-EMP-L where $\theta = \{s_1 := \langle\rangle, s_1' := s\}$. Then there arise the critical pairs:

$$y\theta = (\langle\rangle \ ; \ s_2) \ \otimes \ (s \ ; \ s_2')$$

and

$$x\theta[x_1\theta := v\theta] = s \ ; \ (s_2 \ \otimes \ s_2')$$

Now, because of the precondition to TNF-1, the type of s_2 is $S2 : 0 \Rightarrow n$, but the only possibility for n is 0. Thus, $s_2 = \langle\rangle$. This means that the critical pairs can be rewritten as:

$$y\theta = (\langle\rangle \ ; \ \langle\rangle) \ \otimes \ (s \ ; \ s_2')$$

$$x\theta[x_1\theta := v\theta] = s \;;\; (\langle\rangle \otimes s_2')$$

From here, applications of TNF-EMP and TNF-EMP-L to $y\theta$ reduce it to $s \;;\; s_2'$. Similarly, an application of TNF-EMP-L rewrites $x\theta[x_1\theta := v\theta]$ to the same thing. □

I now show that exhaustive application of the rewrite rules results in a normal form, and that normal form is TNF

Theorem 5 (Structure of the Normal Form)**.** *If $s \nrightarrow s'$ then s is in TNF.*

Proof. By contradiction. Assume that s does not rewrite, but is not in TNF. The proof then proceeds by analysing the structure of s. The only possibilities are:

1. $s \equiv [l]\ s'$
2. $s \equiv s_1 \;;\; s_2$
3. $s \equiv s_1 \otimes s_2$

We then show that in each possible case, a rewrite rule is applicable. For example, consider the second possibility: $s \equiv s_1 \;;\; s_2$. The proof proceeds by considering the possible arities for s_1:

- $s_1 : 0 \Rightarrow m$. The only possibility here is $m = 0$ and $s_1 \equiv s_2 \equiv \langle\rangle$. In this case the rewrite rule TNF-EMP is applicable.
- $s_1 : 1 \Rightarrow m$. In this case, s_1 cannot be the identity tactic or TNF-ID-L would be applicable, nor can it be a tensor. If it is an atomic tactic $s \equiv a \;;\; s_2$, then we need to consider the possible values of m. As before, $m \neq 0$ or the rule TNF-EMP would apply. If $m = 1$ then we have a case analysis on s_2. It cannot be the identity or a tensor. It must then be either a labelled proof $s_2 \equiv [l]\ s_2'$ or a further sequencing $s_2 \equiv s_2' \;;\; s_2''$. For each of these cases, the top-level argument applies.
- $s_1 : n \Rightarrow m$, where $n > 1$. In this case, we again match on the possible structure of s_1. Now it can only be a further sequencing or a tensor $s_1' \otimes s_1''$. In the latter case, we further analyse the possible arities of s_1' and s_1''. In all possible cases, a contradiction can be derived.

□

3.2.4 *Hiproof examples*

Working with abstract tactics like a, b, and c is useful to demonstrate simple hiproofs and demonstrate theoretical properties. To get a real feel for the things, I instantiate the formalism with two derivation systems.

3.2.4.1 *Propositional Logic*

Instantiation of a derivation system (from Definition 1), requires sets $\mathcal{G}$ and $\mathcal{A}$ of goals and atomic tactics. The goals of propositional logic are of the form $\Gamma \vdash P$ where P

is a propositional formula and Γ is a set of propositional formulae. Propositional formulae are given by

Propositional logic

$$P \quad ::= \quad \bot \;\mid\; P \to P \;\mid\; P \wedge P \;\mid\; X$$

where X stands for a propositional variable. Atomic tactics are then given by the well-known rules, of which a sample is given below:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \quad (\textsc{impI})$$

$$\frac{\Gamma \vdash P \qquad \Gamma, Q \vdash R}{\Gamma, P \to Q \vdash R} \quad (\textsc{impE})$$

$$\Gamma, P \vdash P \quad (\textsc{ax})$$

$$\frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R} \quad (\textsc{conjE})$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \quad (\textsc{conjI})$$

Atomic tactics are *instances* of these inference rules, which are viewed as being applied backwards and have the obvious arities.

$$\dfrac{\dfrac{\dfrac{}{P, Q \vdash Q}\,ax}{P \wedge Q \vdash Q}\,conjE \qquad \dfrac{\dfrac{}{P, Q \vdash P}\,ax}{P \wedge Q \vdash P}\,conjE}{\dfrac{P \wedge Q \vdash Q \wedge P}{\vdash P \wedge Q \to Q \wedge P}\,impI}\,conjI$$

Figure 3.14: Derivation tree for $P \wedge Q \to Q \wedge P$

Figure 3.14 shows a natural deduction derivation for the proposition $P \wedge Q \to Q \wedge P$. The *skeleton* proof using the hiproof syntax is:

$$impI \;;\; conjI \;;\; (conjE \;;\; ax) \;\otimes\; (conjE \;;\; ax) \tag{3.11}$$

However, even this simple proof has some structure: it begins with a (short) chain of introduction rules, then each of the subgoals $P \wedge Q \vdash Q$ and $P \wedge Q \vdash P$ are solved by an elimination rule followed by the axiom tactic *ax*. Assuming labels *intros* and *elim* the hiproof representation could be:

$$([intros]\ impI\ ;\ conjI)\ ;\ ([elim]\ conjE\ ;\ ax)\ \otimes\ ([elim]\ conjE\ ;\ ax) \tag{3.12}$$

which, though larger, has the elegant graphical presentation shown in Figure 3.15. The hierarchy could be used to *view* the hiproof simply at the tactic level, as shown in Figure 3.16.

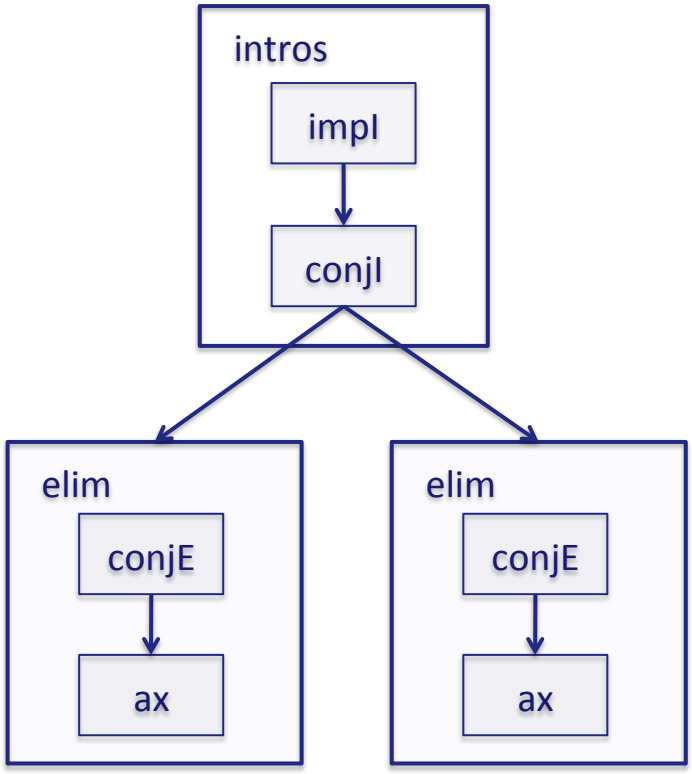

Figure 3.15: Hiproof solving $P \wedge Q \to Q \wedge P$ using two tactics *intros* and *elim*

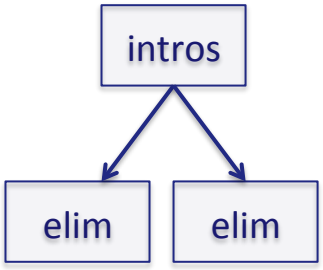

Figure 3.16: Viewing the proof of $P \wedge Q \to Q \wedge P$ at a more abstract level

3.2.4.2 *A simple type theory*

To demonstrate the flexibility of the hiproof notion of proof trees, I provide an example instantiation using the λHOL type theory, which is well-studied in e.g. Barendregt (1992). I start by inductively defining the set of types, $\mathcal{T}$, 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. Thus, a type can be a variable, one of the set of *sorts* ($s = \{\texttt{Prop}, \texttt{Type}, \texttt{Type}'\}$), a type applied to a type ($\mathcal{T}\,\mathcal{T}$), a lambda abstraction ($\lambda\,\mathcal{V} : \mathcal{T}.\,\mathcal{T}$), or a product type ($\Pi\,\mathcal{V} : \mathcal{T}.\mathcal{T}$). Intuitively, lambda abstraction corresponds to function construction and the product type is used to type functions. The ways in which types can be constructed — and correspondingly, the power of the logical system — is defined by typing rules.

The basic construction is a declaration of the form $x : A$ where $A \in \mathcal{T}$ and $x \in \mathcal{V}$, which reads as 'x has type A'. A finite ordered sequence of declarations is called a *context*. Figure 3.17 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.

In the example below, rather than using the Π notation, I use the HOL notation of $\to$, $\forall$, and $\to$ for these three product types. Convertibility of types is represented by $P =_\beta Q$ and defined as the reflexive, transitive, symmetric closure (i.e. equivalence relation) of the standard beta reduction rule:

$$(\lambda x : A.\ B)C \to_\beta B[x := C]$$

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

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

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

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

Figure 3.17: The typing rules for λHOL.

Definition 4 (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.

Figure 3.18 provides a sample set of atomic tactics for this system, as inference rules. As with the propositional logic formulation, they 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. I describe some of the atomic tactics:

$$\frac{\Gamma \vdash t : P}{\Gamma \vdash P} \quad \text{(A-Exact}(t)\text{)}$$

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

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

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

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

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

Figure 3.18: Atomic tactics

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

A-ASSUMPTION A declaration in the context with a type convertible with the goal can solve the goal using the tactic *assumption*.

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-REFINE(t) Refinement with the *refine* tactic 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.

Atomic tactics preserve *well-formedness* in the sense that given a well-formed goal (Definition 4), application of the tactic produces more well-formed goals. To be completely formal, would require showing that the atomic tactics are sound with respect to the low-level rules of λHOL. That is, if a proof term which inhabits the type of each of the subgoals can be constructed, then so can a proof term inhabiting the type of the original goal. As a first example proof in this framework, consider the type:

$$\forall A\ B\ C : \texttt{Prop}.\ (A \to B \to C) \to (A \to B) \to A \to C$$

which is proved in Figure 3.19. The derivation tree is rather large, but the first six steps are all bookkeeping, and the interesting stuff doesn't happen until the *refinement*. The first refinement applies a term $ABC\ ?x\ ?y$ which is of type C but includes two schematic variables $?x : A$ and $?y : B$. The refinement tactics substitutes these as the new subgoals. The first, A, is easy to solve as it exists in the context. Solving B needs another refinement. This time, applying the term $AB\ ?x$ (of type B) reduces the goal

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots, hA : A \vdash A}\,assumption}{\ldots, AB : A \to B,\ hA : A \vdash B}\,refine(AB\ ?x)}{\ldots, ABC : (A \to B \to C),\ AB : (A \to B),\ hA : A \vdash C}\,refine(ABC\ ?x\ ?y))}{\ldots, ABC : (A \to B \to C),\ AB : (A \to B) \vdash A \to C}\,intro(hA)}{\ldots, ABC : (A \to B \to C) \vdash (A \to B) \to A \to C}\,intro(AB)}{A,\ B,\ C : \texttt{Prop} \vdash (A \to B \to C) \to (A \to B) \to A \to C}\,intro(ABC)}{A,\ B : \texttt{Prop} \vdash \forall C : \texttt{Prop}.\ (A \to B \to C) \to (A \to B) \to A \to C}\,intro(C)}{A : \texttt{Prop} \vdash \forall B\ C : \texttt{Prop}.\ (A \to B \to C) \to (A \to B) \to A \to C}\,intro(B)}{\vdash \forall A\ B\ C : \texttt{Prop}.\ (A \to B \to C) \to (A \to B) \to A \to C}\,intro(A)$$

Figure 3.19: A derivation tree proving $\forall A\ B\ C : \texttt{Prop}.\ (A \to B \to C) \to (A \to B) \to A \to C$

to solving A again, which is in the context. One can view the overall structure of the proof using a hiproof like that in Figure 3.20, which separates the bookkeeping steps dealing with quantifications and assumptions. The advantage of hiproofs becomes more evident when viewed at the level of abstraction that hides the details of the quantifier elimination and assumption introduction phases, Figure 3.21, making the proof more compact and understandable[5].

3.3 HIERARCHICAL TACTICS

Constructing proofs directly from inference rules is a tedious process. I now introduce the Hitac language to facilitate proof search and the use of lemmas.

3.3.1 *A Hitac term language*

Figure 3.22 introduces the syntax for Hitac. Hitac extends hiproofs with five additional constructs. The first four are designed to allow proof search:

GOAL ASSERTION ($assert\ \gamma$) allows the control flow to be adjusted, by matching the current goal against the supplied parameter goal (γ).

ALTERNATION ($t \mid t$) allows alternative proof branches to be followed.

DEFINED TACTICS ($name(t, \ldots, t)$) allow an environment of possibly mutually recursive tactics to be constructed.

TACTIC VARIABLES (X) allow the environment to contain parameterised tactics, such as $REPEAT(X)$, often called tacticals.

The final Hitac construct ($lem\ l$) extends the language given in Aspinall et al. (2010) with *lemma applications*, which allow an environment of previously proved facts to be utilised by the tactic language, saving repeated proof search. The conventions from hiproofs are extended, where alternation has the lowest precedence, then sequencing,

5 Note, of course that given a sophisticated hiproof viewer, the goals would be visible as well, providing more details about the proof.

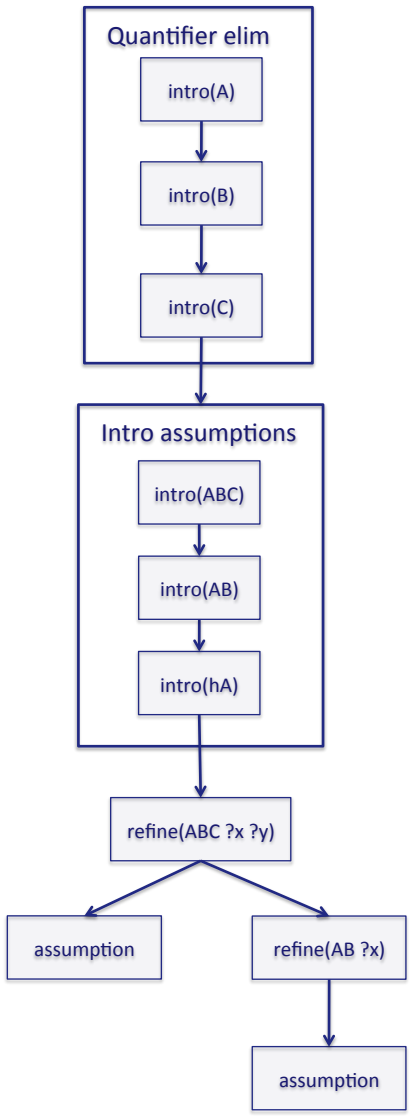

Figure 3.20: Viewing the proof of $\forall A\ B\ C$: `Prop`. $(A \to B \to C) \to (A \to B) \to A \to C$ as a hiproof

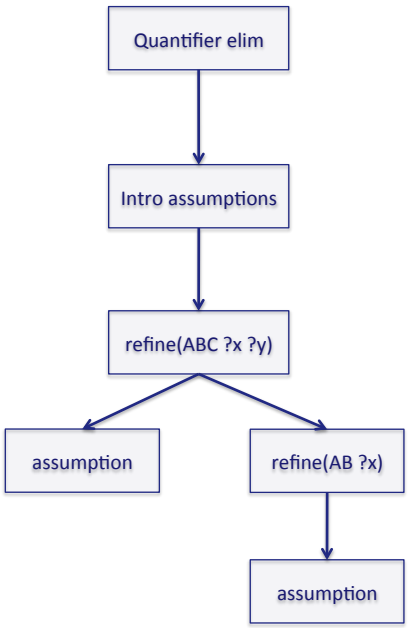

Figure 3.21: Abstracted view of the hiproof of $\forall A\ B\ C$: `Prop`. $(A \to B \to C) \to (A \to B) \to A \to C$

Hitac syntax

t	$::=$	a	an atomic tactic
	$\mid$	id	identity tactic
	$\mid$	$[l]\ t$	labelled tactic
	$\mid$	$t\ ;\ t$	sequential composition of tactics
	$\mid$	$t \otimes t$	parallel composition of tactics: 'tensor'
	$\mid$	$\langle\rangle$	empty
	$\mid$	$swap$	goal swapping
	$\mid$	$assert\ \gamma$	goal assertion
	$\mid$	$t \mid t$	alternation
	$\mid$	$name(t, \ldots, t)$	defined tactic, with parameters
	$\mid$	X	tactic variables
	$\mid$	$lem\ l$	lemma applications

Figure 3.22: Linear syntax for hierarchical tactics

then tensor. Any defined tactics that do not have parameters will be written without brackets. I will often write $\overline{X}$ or $\overline{t}$ for lists of tactic variables and tactics respectively. Substitution for tactic variables is defined as follows.

Definition 5 (Tactic variable substitution)**.** For a tactic variable X and two tactics t and t', the substitution of t' for X in t is written as $t[X := t']$. Multiple substitutions are written $t[\overline{X} := \overline{t'}]$.

The set of available lemmas and defined tactics available for use in a tactic term is defined by a *proof environment*.

Definition 6 (Lemma environment)**.** A *lemma environment* $\mathcal{L}$ is a map

$$\mathcal{L} : \mathit{name} \to (\mathit{goal} \times s)$$

consisting of the lemma name, the goal it solves, and the proof object (a hiproof). The map must be *well-formed*. That is, for every *name*, if $\mathcal{L}(\mathit{name}) = (\gamma, s)$ then:

$$s \vdash \gamma \longrightarrow g$$

for some g. The lemma may leave danglers, i.e., introduce subgoals. These can be thought of as the assumptions for the lemma.

Definition 7 (Tactic environment)**.** A *tactic environment* $\mathcal{T}$ is a well-formed map:

$$\mathcal{T} : \mathit{name} \to (\mathit{var\ list}) \times t$$

sending a tactic name to the relevant list of parameters, and the hitac. I will often abuse notation and write $\mathit{mytac} \in \mathcal{T}$ to mean that there is a tactic called *mytac* in the environment.

Tactic environments are also subject to well-formedness conditions; however, since tactics can contain lemma applications, their well-formedness is with respect to a given lemma environment. The precise definition of well-formedness is postponed until Section 3.3.3, where *well-formedness* checking for tactics is introduced, which characterises when an individual tactic is well-formed with respect to a given environment.

Definition 8 (Proof environment)**.** A *proof environment* is a pair $(\mathcal{T}, \mathcal{L})$ of well-formed tactic and lemma environments.

3.3.2 *Big step evaluation semantics*

Tactics are executed against a list of goals resulting in a (possibly empty) list of remaining subgoals and a proof object: a hiproof. Hitac evaluation is defined with the relation:

$$\langle g, t \rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, g' \rangle \tag{3.13}$$

which can be read as 'the tactic t applied to the proof context g returns a hiproof s and updated context g', under the proof environment $(\mathcal{T}, \mathcal{L})'$. In the evaluation rules,

$$\frac{\frac{\gamma_1, \ldots, \gamma_n}{\gamma}\ a \in \mathcal{A}}{\langle [\gamma], a\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle a, [\gamma_1, \ldots, \gamma_n]\rangle} \quad \text{(B-Tac-Atomic)}$$

$$\langle [\gamma], id\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle id, \gamma\rangle \quad \text{(B-Tac-Id)}$$

$$\frac{\langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, [l]\ t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [l]\ s, g\rangle} \quad \text{(B-Tac-Lab)}$$

$$\frac{\langle g_1, t_1\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, g_2\rangle \qquad \langle g_2, t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_2, g_3\rangle}{\langle g_1, t_1\ ;\ t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1\ ;\ s_2, g_3\rangle} \quad \text{(B-Tac-Seq)}$$

$$\frac{\langle g_1, t_1\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, g_1'\rangle \qquad \langle g_2, t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_2, g_2'\rangle}{\langle g_1\ @\ g_2, t_1\ \otimes\ t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1\ \otimes\ s_2, g_1'\ @\ g_2'\rangle} \quad \text{(B-Tac-Tens)}$$

$$\langle [], \langle\rangle\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle \langle\rangle, []\rangle \quad \text{(B-Tac-Empty)}$$

$$\langle [\gamma_1, \gamma_2], swap\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle swap, [\gamma_2, \gamma_1]\rangle \quad \text{(B-Tac-Swap)}$$

$$\frac{\gamma\ \sim\ \gamma'}{\langle [\gamma], assert\ \gamma'\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle id, [\gamma]\rangle} \quad \text{(B-Tac-Assert)}$$

$$\frac{\langle g, t_1\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle} \quad \text{(B-Tac-Alt-L)}$$

$$\frac{\langle g, t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle}{\langle g, t_1 \mid t_2\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle} \quad \text{(B-Tac-Alt-R)}$$

$$\frac{\mathcal{T}(name) = (\overline{X}, t) \qquad \langle g, t[\overline{X} := \overline{t}]\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle}{\langle g, name(\overline{t})\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g'\rangle} \quad \text{(B-Tac-Def)}$$

$$\frac{\mathcal{L}(l) = (\gamma, s) \qquad s \vdash \gamma \longrightarrow g}{\langle [\gamma], lem\ l\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [lem\ l]\ s, g\rangle} \quad \text{(B-Tac-Lem)}$$

Figure 3.23: Hitac big step evaluation rules

I will contract the proof environment $(\mathcal{T}, \mathcal{L})$ and write it simply as $\mathcal{E}$ for brevity. A tactic t will then *prove* a goal γ if

$$\langle [\gamma], t \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, [] \rangle$$

for some hiproof s. The tactic evaluation relation is defined inductively by the rules in Figure 3.23.

The big step rules capture the intended meaning of each tactic. For example, the rule B-Tac-Seq evaluates a tactic $t_1 \; ; \; t_2$ on a list of goals g_1 by applying the first tactic t_1 resulting in a list of remaining goals g_2 (and hiproof s_1). The new proof context is then passed to the second tactic t_2, which is evaluated to obtain g_3 and another hiproof s_2. The resulting hiproof is constructed by composing the individual proofs.

An assertion succeeds with the identity hiproof if the current goal matches the asserted goal with respect to $\sim$, the matching relation on the derivation system. If they do not match, tactic evaluation will fail. Note that there is no explicit rule for tactic variable evaluation: variables must be fully substituted before tactic evaluation can succeed. There are two rules for alternation, allowing a non-deterministic choice.

The evaluation of defined tactics proceeds as might be expected. The rule first checks that the tactic exists in the tactic environment before substituting the formal parameters for actual parameters and evaluating the resulting tactic. If a definition with n arguments is not applied to n arguments, then it simply fails to reduce. Similarly, if a definition contains a tactic variable not present in the formal parameters, then it will also fail to reduce. Lemma application means simply picking the appropriate lemma from the environment and checking that it validates the current goal. Interestingly, this rule also introduces a labelled box to allow tracking of the lemma application (since otherwise, all that would be visible is the hiproof of that lemma).

Theorem 6 (Soundness of big step evaluation rules)**.** *If* $\langle g, t \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g' \rangle$ *then* $s \vdash g \longrightarrow g'$.

Proof. The proof proceeds by a straightforward induction on the derivation of $\langle g, t \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g' \rangle$. The rules B-Tac-Atomic to B-Tac-Swap all match directly with the validation rules V-Atomic to V-Swap. For the rules B-Tac-Alt-L and B-Tac-Alt-L, the resultant hiproof s appears in the precondition, so the induction hypothesis can be used directly. For B-Tac-Assert, use the validation rule V-Id. Finally, for B-Tac-Lem, V-Lab combined with the direct hiproof validation for the lemma give the result. □

3.3.3 *Well-formed tactics*

Given a proof environment $(\mathcal{T}, \mathcal{L})$, I define a judgment to ensure that any given tactic t is well-formed with respect to this environment:

$$(\mathcal{T}, \mathcal{L}) \vdash t$$

That is, any tactics and lemmas used by t have been defined/proved (they exist in the environment) and are used correctly. Figure 3.24 gives the inductive rules that define this judgement (where I contract the proof environment $(\mathcal{T}, \mathcal{L})$ and write it simply as $\mathcal{E}$ for brevity). The atomic tactics, the identity tactic, empty tactic, tactic variables and the swap tactic are always well-formed. Each of the binary operations, for example

$$\frac{a \in \mathcal{A}}{\mathcal{E} \vdash a} \quad \text{(TT-Atomic)}$$

$$\mathcal{E} \vdash id \quad \text{(TT-Id)}$$

$$\frac{\mathcal{E} \vdash s}{\mathcal{E} \vdash [l]\ s} \quad \text{(TT-Lab)}$$

$$\frac{\mathcal{E} \vdash s_1 \qquad \mathcal{E} \vdash s_2}{\mathcal{E} \vdash s_1 \ ;\ s_2} \quad \text{(TT-Seq)}$$

$$\frac{\mathcal{E} \vdash s_1 \qquad \mathcal{E} \vdash s_2}{\mathcal{E} \vdash s_1 \ \otimes\ s_2} \quad \text{(TT-Tens)}$$

$$\mathcal{E} \vdash \langle\rangle \quad \text{(TT-Empty)}$$

$$\mathcal{E} \vdash swap \quad \text{(TT-Swap)}$$

$$\frac{\gamma \in \mathcal{G}}{\mathcal{E} \vdash assert\ \gamma} \quad \text{(TT-Assert)}$$

$$\frac{\mathcal{E} \vdash s_1 \qquad \mathcal{E} \vdash s_2}{\mathcal{E} \vdash s_1 \mid s_2} \quad \text{(TT-Alt)}$$

$$\frac{\mathcal{T}(name) = (\overline{X}, t) \qquad len(\overline{X}) = n \qquad \mathcal{E} \vdash t_1 \ldots \mathcal{E} \vdash t_n}{\mathcal{E} \vdash name(t_1, \ldots, t_n)} \quad \text{(TT-Def)}$$

$$\mathcal{E} \vdash X \quad \text{(TT-Var)}$$

$$\frac{l \in \mathcal{L}}{\mathcal{E} \vdash lem\ l} \quad \text{(TT-Lem)}$$

Figure 3.24: Tactic well-formedness rules

a sequential composition, are well-formed if the left-hand side and right-hand side are both well-formed. Similarly, a labelled tactic is well-formed if the tactic being labelled is well-formed and assertions are well-formed if the goal being asserted is well-formed. Checking a defined tactic amounts to ensuring that it exists in the tactic environment and that it has the correct number of parameters, each of which are well-formed. Finally, a lemma application is checked by ensuring the lemma name exists in the environment.

This *static check* on hitacs is useful to remove some sources of failure in tactic execution. It cannot, however, ensure that a well-formed tactic will evaluate without failure. This is due to the dependence of a tactic behaviour on the goal it will be applied to: assertions may fail, tactics may recurse, alternative paths may be chosen. All this means that, unlike hiproofs, a fixed arity cannot be attached to a hitac.

$$\mathcal{L} \vdash \{\} \qquad \text{(TE-Empty)}$$

$$\frac{\mathcal{L} \vdash \mathcal{T} \qquad \mathit{variables}(t) \subset \overline{X} \qquad (\mathcal{T}, \mathcal{L}) \vdash t}{\mathcal{L} \vdash \mathcal{T}[\mathit{name} \mapsto (\overline{X}, t)]} \qquad \text{(TE-Cons)}$$

Figure 3.25: Tactic environment well-formedness rules

Tactic checking is also useful for ensuring well-formedness of tactic environments, represented by the judgement:

$$\mathcal{L} \vdash \mathcal{T} \tag{3.14}$$

and defined in Figure 3.25. This judgement provides the appropriate well-formedness criterion for Definition 7.

3.3.4 *Minimal environments*

Minimal environments answer the question:

> 'what is the minimal amount of stuff needed in a proof environment for this to be a well-formed tactic?'

Firstly, I write $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}, \mathcal{L})$ to mean that $\mathcal{T}' \subset \mathcal{T}$ and/or $\mathcal{L}' \subset \mathcal{L}$ (with the maps viewed as sets), and $(\mathcal{T}', \mathcal{L}') \subseteq (\mathcal{T}, \mathcal{L})$ is the reflexive closure. Then:

Definition 9 (Minimal environment)**.** A proof environment $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$ is a *minimal environment* for a tactic t if:

1. $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}}) \vdash t$;
2. For every well-formed proof environment $(\mathcal{T}', \mathcal{L}')$ such that $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$, the tactic is not well-formed.

That is: take anything away from a minimal environment and it will fail to judge the tactic well-formed.

Given an environment $(\mathcal{T}, \mathcal{L})$ and a tactic t (well-formed under $(\mathcal{T}, \mathcal{L})$), define the *t-restricted environment* $(\mathcal{T}|_t, \mathcal{L}|_t)$ as follows:

$$\begin{aligned} \mathcal{T}|_t &= \{\, (n, \mathcal{T}(n)) \mid n \in \mathit{tacs}(t) \,\} \\ \mathcal{L}|_t &= \{\, (n, \mathcal{L}(n)) \mid n \in \mathit{lemmas}(t) \,\} \end{aligned}$$

where *tacs* and *lemmas* are defined inductively on the structure of hitac tactics and contain the set of names of tactics $\mathfrak{n}$ (or lemmas $\mathfrak{l}$) appearing in t or transitively in the body of any defined tactics.

It is easy to see that $(\mathcal{T}|_t, \mathcal{L}|_t)$ is a well-formed environment; furthermore, it is minimal:

Theorem 7 (Minimality of $(\mathcal{T}|_t, \mathcal{L}|_t)$)**.** *Given an initial environment $(\mathcal{T}, \mathcal{L})$ and a tactic t, the environment $(\mathcal{T}|_t, \mathcal{L}|_t)$ defined as above is a minimal environment.*

Proof. To prove this, we must show:

1. $(\mathcal{T}\big|_t, \mathcal{L}\big|_t) \vdash t$. This is established by induction on the well-formedness of the tactic in the original environment. Thus, for the derivation below:

$$\frac{\mathcal{T}(name) = (\overline{X}, t) \qquad len(\overline{X}) = n \qquad (\mathcal{T}, \mathcal{L}) \vdash t_1 \ldots (\mathcal{T}, \mathcal{L}) \vdash t_n}{(\mathcal{T}, \mathcal{L}) \vdash name(t_1, \ldots, t_n)}$$

   We know that $name \in tacs(t)$ and, by the definition of $\mathcal{T}\big|_t$, that $\mathcal{T}\big|_t(name) = \mathcal{T}(name) = (\overline{X}, t)$ and so (with the final premises holding by the induction hypothesis):

$$\frac{\mathcal{T}\big|_t(name) = (\overline{X}, t) \qquad len(\overline{X}) = n \qquad (\mathcal{T}\big|_t, \mathcal{L}\big|_t) \vdash t_1 \ldots (\mathcal{T}\big|_t, \mathcal{L}\big|_t) \vdash t_n}{(\mathcal{T}\big|_t, \mathcal{L}\big|_t) \vdash name(t_1, \ldots, t_n)}$$

   The lemma rule is similar, and the others are straightforward.

2. We then need to show that for every $\mathcal{T}', \mathcal{L}'$ such that $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}\big|_t, \mathcal{L}\big|_t)$, the tactic is not well-formed. The basic strategy is similar: argue that at some point in the derivation, there must be a rule where $\mathcal{T}'(name)$ is undefined and thus well-formedness checking fails.

□

Theorem 8 (Closure of well-formedness under environment extension)**.** *If* $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}, \mathcal{L})$ *and* $(\mathcal{T}', \mathcal{L}') \vdash t$ *then* $(\mathcal{T}, \mathcal{L}) \vdash t$.

Proof. The proof of this is a straightforward induction on the evaluation. □

3.3.5 *Examples*

To get a better feeling for how hitacs can be written, how they evaluate and how they construct hiproofs, it's best to start building and evaluating them. I follow the LCF tradition and call parameterised tactics *tacticals*. I also adopt the LCF convention that tactics and tacticals are written in uppercase. Again, most of this material is drawn from Aspinall et al. (2010).

3.3.5.1 *Building LCF from hitac*

The LCF theorem prover was the first system to introduce the notion of tactics and tacticals (Gordon et al., 1978). The system came with a set of built-in tacticals that were frequently used to compose tactics together in different ways. In this section, I show how Hitac can model the LCF tacticals. Each LCF tactic operates on a single goal and returns a list of subgoals. Since hitac tactics can be more general, I often use the identity tactic *id* to force evaluation to fail if the hitac is applied to more than one goal.

T_1 *THEN* T_2 The classic *THEN* tactical operates on a single goal and applies T_1 to that goal. It then applies T_2 to all the subgoals generated from the application of T_1. To represent this in hitac, a paramaterised tactic (a tactical) that applies its parameter to all subgoals is needed:

$$ALL(X) := (id\ ;\ X)\ \otimes\ (ALL(X) \mid \langle\rangle) \tag{3.15}$$

When confronted with a list of goals, this tactic applies X to the first goal (forced by the identity tactic); to the remaining subgoals, it applies the alternatives $(ALL(X) \mid \langle\rangle)$. If the goal list is now empty — that is, it was a singleton list to start with — then the only alternative that can succeed is the empty tactic; otherwise, the $ALL(X)$ is called again. This time, though, the list of goals is shorter: thus, recursion must end.

Using this, a more general identity tactic that, instead of operating only on a single goal, does nothing to a whole list of goals can be defined:

$$ID := ALL(id) \mid \langle\rangle \tag{3.16}$$

which behaves as expected and is also applicable to an empty list of goals. Interestingly, when applying ID to a list of goals $[\gamma_1, \gamma_2, \gamma_3]$ the resulting hiproof is:

$$(id\ ;\ id)\ \otimes\ (id\ ;\ id)\ \otimes\ (id\ ;\ id)\ \otimes\ \langle\rangle$$

because of the 'arity checking' *id* in hitac 3.15. A little bit of hiproof normalisation gives a neater proof:

$$id\ \otimes\ id\ \otimes\ id$$

Using *ALL*, define:

$$THEN(t_1, t_2) := id\ ;\ t_1\ ;\ ALL(t_2) \tag{3.17}$$

where the *id* tactic is used to ensure this tactical operates on a single goal. The $ALL(t_2)$ will then apply t_2 to all the goals generated by t_1.

$REPEAT(T)$ Repetition is an important tactical to shorten proof scripts. Consider, for example, an *intros* tactic: it repeatedly tries to apply introduction rules until all assumptions, quantifiers etc. have been stripped from a formula. The LCF *REPEAT* tactical does this. It is defined as follows:

$$REPEAT(X) := id\ ;\ X\ ;\ ALL(REPEAT(X)) \mid ID \tag{3.18}$$

noting that alternation has lower precedence than sequencing. Thus, this tactic reads as: 'do X ; $ALL(REPEAT(X))$ or else ID'.

To get a good feeling for this tactic, imagine applying $REPEAT(conjI)$ to the goal $\mathsf{P} \wedge (\mathsf{Q} \wedge \mathsf{R})$. After unfolding the definition and applying the first side of the sequential composition, there are two subgoals $[\mathsf{P}, \mathsf{Q} \wedge \mathsf{R}]$. Using ALL means $REPEAT(conjI)$ is applied to each subgoal. Attempting to apply $conjI$ to the subgoal P fails; thus, the whole sequencing fails and the right hand side of the alternation operator gets applied i.e. the general identity, which stops evaluation on this goal. The tactic has better luck on the second subgoal $\mathsf{Q} \wedge \mathsf{R}$ and can apply `conjI` again to get $[\mathsf{Q}, \mathsf{R}]$ as the

new list of subgoals. Again, ALL is applied and recurses REPEAT on each subgoal before failing and applying the general identity.

The resulting hiproof term is quite complicated after all the interaction between the recursive tacticals (remembering that ID also uses ALL), but after normalisation the hiproof is simply:

$$conjI\ ;\ (id \otimes conjI)$$

T $THENL$ $[T_1, \ldots, T_n]$ Finally, the LCF '*THEN LIST*' tactical behaves like $THEN$ except that as a second parameter, it takes a list of tactics equal in length to the subgoals generated by the first tactic. Hitac cannot, in fact, represent this generically as it does not contain any list data structure. However, any instance of $THENL$ can be intuitively represented in hitac:

$$t\ THENL\ [t_1, \ldots, t_n] := id\ ;\ t\ ;\ (t_1 \otimes \ldots \otimes t_n) \tag{3.19}$$

T_1 $ORELSE$ T_2 This tactic interestingly cannot be modelled directly by the alternation operator of hitac as it introduces non-determinism in the choose of which tactic is applied whereas in LCF T_1 is always tried first.

3.3.5.2 *Changing the order of goals*

It can be helpful to modify the order of goals; either to solve the simple goals first or to group similar goals together. In hitac, this is achieved by utilising the *swap* tactic. The raw swapping tactic, however, is too low-level: only swapping the order of two adjacent subgoals. This section demonstrates how to build *rotation* and *reflection* tactics. Before defining these tactics, I first define a handy utility tactic $NULL$ that succeeds (and does nothing) when given a singleton goal list or an empty goal list:

$$NULL := \langle\rangle \mid id \tag{3.20}$$

$ROTATE_L$ This tactic will shift the order of goals to the left by one each time it is applied. Thus, the list $[\gamma_1, \gamma_2, \gamma_3, \gamma_4]$ will be transformed to $[\gamma_2, \gamma_3, \gamma_4, \gamma_1]$. This is defined as:

$$ROTATE_L := [(swap \otimes ID)\ ;\ (id \otimes ROTATE_L)] \mid NULL \tag{3.21}$$

The base case for the recursion is the top level alternation, where $NULL$ deals with a singleton or empty list, which are identical when rotated. For larger lists it first applies the left hand side of the sequencing $(swap \otimes ID)$. This swaps the first two goals and applies the general identity to the remaining. Thus, a list $[\gamma_1, \gamma_2, \gamma_3, \gamma_4]$ will be mapped to $[\gamma_2, \gamma_1, \gamma_3, \gamma_4]$ and a (normalised) hiproof $swap \otimes id \otimes id \otimes \langle\rangle$ is constructed. Note now that the first goal γ_2 is in the right position. This motivates the right hand side of the sequencing $(id \otimes ROTATE_L)$ as the recursive call is only made to the list $[\gamma_1, \gamma_3, \gamma_4]$. Thus, another iteration of the left hand side hitac results in the goal list $[\gamma_3, \gamma_1, \gamma_4]$; and now γ_3 will be in the correct position when the tensored goal lists are composed. Again the recursion is applied to the reduced list $[\gamma_1, \gamma_4]$ which swaps these two (with the general identity tactic operating on the empty list, which is fine according to the definition as hitac 3.16). At this point, the

next recursive call needs to apply the *null* tactic, which halts the tactic. The tensors are composed to end up with the goal list required and a (normalised) hiproof as follows:

$$\begin{aligned} &swap \otimes id \otimes id\ ; \\ id \otimes\ &[(swap \otimes id)\ ; \\ &id \otimes (swap)] \end{aligned}$$

$ROTATE_R$ Similarly to $ROTATE_L$, $ROTATE_R$ will shift the order of goals to the right by one. Thus, the list $[\gamma_1, \gamma_2, \gamma_3, \gamma_4]$ will be transformed to $[\gamma_4, \gamma_1, \gamma_2, \gamma_3]$. The definition is similar to hitac 3.21:

$$ROTATE_R := [(ID \otimes swap)\ ;\ (ROTATE_R \otimes id)] \mid NULL \tag{3.22}$$

except that the swaps propagate from the end of the list.

$REFLECT$ Reflection will invert the order of subgoals, transforming the list of goals $[\gamma_1, \gamma_2, \gamma_3, \gamma_4]$ to $[\gamma_4, \gamma_3, \gamma_2, \gamma_1]$. In order to define reflection, I observe that an application of $ROTATE_R$ gets the first goal in the right place (i.e. γ_4 becomes the first goal); then, if repeatedly applied to the tail of the list will reflect the list:

$$ROTATE_R\ ;\ (id \otimes REFLECT) \mid NULL \tag{3.23}$$

3.4 SUMMARY

Hierarchical proofs and tactics feature heavily in this thesis and I hope this chapter has done much to give a good feeling for the nature and behaviour of the term languages introduced as well as the graphical presentation lurking in the background. While I do not focus on it, the presentation of hiproofs in a graphical viewer and/or editor is an important part of the allure of such a representation for proofs.

Moreover, this thesis contributes to the understanding of hierarchical proof in several ways: I study the swapping of goal order — visualised as the crossing of goal edges — and normalisation of hierarchical proofs: the reduction of different terms that are semantically identical to a syntactic normal form. Furthermore the hitac language was extended to include *lemma applications* that can reduce the need for repeated proof search and show that it is still correct. I introduced a concept of static well-formedness checking for hitacs that acts as a very simple typing system that removes a certain class of evaluation failures and concluded this chapter with a demonstration of the Hitac language for constructing the tacticals of the LCF system as well as high-level goal reordering tactics, both of which are used in forthcoming chapters.

4

THE HISCRIPT PROOF LANGUAGE

4.1 INTRODUCTION

Despite the growing number and increased use of declarative languages, there has been comparatively little investigation into formal semantics of proof languages. Providing formal semantics for proof languages is an important first step on the path to properly understanding the nature of proof languages and enables us to reason about proof in a similar way to programming languages. This chapter takes some steps in this direction and describes a formal semantics for a declarative proof language called Hiscript. Hiscript is an experimental, generic declarative proof language, similar to Isar. I use Hitac as the underlying tactic language for Hiscript and provide (and prove correct) a big step evaluation semantics for it that constructs hiproofs.

CHAPTER MAP In Sections 4.2 and 4.3 I describe the syntax and semantics of the language and throughout the chapter, examples of the language in action are provided in order to get a feeling for the styles of proof available. Section 4.5 extends the well-formedness checking for tactics to declarative proofs and provides and precise notion of dependencies. I discuss some of the features of Hiscript and compare the language with other declarative proof languages in Section 4.6.

CONTRIBUTIONS This chapter contributes an investigation into semantics for the declarative proof language Hiscript. To be more precise:

1. The proof language Hiscript is *generic*: it does not prescribe an underlying logic; and it is hierarchical: providing explicit (and also implicit) hierarchical constructs.
2. The evaluation semantics for Hiscript is proved correct in the technical sense that it constructs valid hiproofs.
3. Well-formedness checking rules for Hiscript are shown to trap potential errors before evaluation.
4. A precise notion of *gaps* in Hiscript proofs is given.

An earlier version of this proof language was presented in Whiteside et al. (2011).

4.2 THE HISCRIPT PROOF LANGUAGE

The following declarative proof language is experimental and lacks features from real languages. The most obvious omission from the language is explicit constructs for stating and manipulating assumptions: a casualty of the generic approach. Proofs in Hiscript are ranged over by the *prf* meta-variable and are constructed by the grammar given in Figure 4.1.

Hiscript syntax

rule	::=	*t*	Proof rules are hitacs.
prf	::=	**proof**(*rule*) *stmt** **qed**	A proof block starts with a rule contains zero or more statements and finishes with a qed.
	|	**gap**	Gaps allow incomplete proofs.
	|	[*name*] *prf*	Labelling proofs is possible.
stmt	::=	**apply** *rule*	Statements can be procedural steps;
	|	**show** (*name* :)$^?$ *goal* *prf*	solve a stated goal;
	|	**have** *name* : *goal* *prf*	introduce a new local lemma;
	|	**tac** *name*($\overline{X}$) := *rule*	introduce a new local proof rule;
	|	**from** *name** **show** *name* : *goal* **by** *rule*	forward step using list of lemmas;
	|	**from** *name** **have** *name* : *goal* **by** *rule*	forward adding local lemma.

Figure 4.1: Syntax for declarative proofs

The core component of the language is a *proof block*:

proof(*rule*)

$stmt_1$
$\vdots$
$stmt_n$
qed

Proof blocks operate on a single goal, applying the initial rule — called a proof block introduction tactic — before solving the resulting subgoals using the statements inside it. If the initial rule introduces n subgoals, then they must all be solved inside the block. If the initial rule solves its goal — that is, there are no statements inside — **by** *rule* can be used as a syntactic convenience. That is:

$$\textbf{by}\ rule \equiv \textbf{proof}(rule)\ \textbf{qed}$$

The other type of *prf* is known as a **gap**. Gaps provide a 'dummy' proof of a single goal, similar to the **sorry** command of Isar. Explicit hierarchical boxes can be added to proof blocks using the labelling construct $[name]\ prf$.

The key statement for making progress in a proof is **show**, which allows the user to supply a proof to the next goal. The user can optionally supply a meaningful name for the goal (which is also used to label the proof in the hierarchy constructed in the underlying hiproof). Tactics can be applied directly using the **apply** statement. The language supports a forward proof style by using the **have** statement to introduce local lemmas that extend the environment, then

from $lem_1 \ldots lem_n$ **show** *goal* **by** *rule*

to perform the step. Finally, in analogy with local lemmas, local tactics can be defined using the **tac** statement. These tactics can have parameters, which must be instantiated when used.

The following examples show Hiscript in action.

Example 1 (Propositional Logic)**.** Firstly, using the logic defined in Section 3.2.4, two proofs of the goal:

$$(P \to Q \to R) \to (P \wedge Q) \to R \tag{4.1}$$

are given. The first proof, shown in Listing 4.1 uses only the basic inference rules of the language and uses only the backwards constructs.

If I define a few simple tactics, they can shorten the proof, as seen in Listing 4.2. The language allows the user flexibility in choosing what details of the proof to present.

Example 2 (Set Theory)**.** Listings 4.3 and 4.4 present two example proofs in the logic SET, for which some of the atomic tactics are given in Figure 4.2.

Listing 4.3 presents the detailed proof and Listing 4.4 presents a more compact, forwards style proof. The defined tactic *intro* will first try to apply an introduction rule for subsets, then the introduction rule for intersection; if both of these fail, then the tactic itself fails.

```
proof(impl)
  show P → Q → R ⊢ P ∧ Q → R
  proof(impl)
    show P → Q → R, P ∧ Q ⊢ R
    proof(mp)
      show P ∧ Q ⊢ Q by conjE ; ax
      show P → Q → R, P ∧ Q ⊢ Q → R
      proof (mp)
        show P ∧ Q ⊢ P by conjE ; ax
        show P → Q → R, P ∧ Q ⊢ P → Q → R by ax
      qed
    qed
  qed
qed
```

Listing 4.1: A proof of $(P \to Q \to R) \to (P \wedge Q) \to R$

```
proof
 tac intros := impl ; impl
 tac conjAx := conjE ; ax
 show (P → Q → R) → P ∧ Q → R
  proof( intros )
    show P → Q → R, P ∧ Q ⊢ R
    proof(mp)
      show P ∧ Q ⊢ Q
        by conjAx
     show P → Q → R, P ∧ Q ⊢ Q → R
       by (mp ; conjAx ⊗ ax)
    qed
  qed
qed
```

Listing 4.2: A proof of $(P \to Q \to R) \to (P \wedge Q) \to R$ using tactics

$$\frac{\Gamma \vdash x \in A \qquad \Gamma \vdash x \in B}{\Gamma \vdash x \in A \cap B} \quad (\cap\text{-INTRO})$$

$$\frac{\Gamma, x \in A \vdash x \in B}{\Gamma \vdash A \subset B} \quad (\subset\text{-INTRO})$$

$$\frac{\Gamma, x \in A, x \in B \vdash C}{\Gamma, x \in A \cap B \vdash C} \quad (\cap\text{-ELIM})$$

Figure 4.2: Some atomic tactics for a SET theory

```
proof
  tac intro := ⊂—intro | ∩—intro
  show A ∩ B ⊂ B ∩ A
  proof( intro )
    show x ∈ A ∩ B ⊢ x ∈ B ∩ A
    proof( intro )
      show x ∈ A ∩ B ⊢ x ∈ B by ∩—elim ; ax
      show x ∈ A ∩ B ⊢ x ∈ A by ∩—elim ; ax
    qed
  qed
qed
```

Listing 4.3: A proof of $A \cap B \subset B \cap A$

```
proof
  tac intro := ⊂—intro | ∩—intro
  have a: x ∈ A ∩ B ⊢ x ∈ A by ∩—elim ; ax
  have b:  x ∈ A ∩ B ⊢ x ∈ B by ∩—elim ; ax
  from b a show A ∩ B ⊂ B ∩ A by intro ; intro
qed
```

Listing 4.4: A proof of $A \cap B \subset B \cap A$ using a forward proof style

4.3 HISCRIPT SEMANTICS

Evaluation of a *prf* is given by the following relation:

$$\langle g, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle \tag{4.2}$$

which can be read as 'the proof *prf* is a Hiscript proof of the goals g under the environment $(\mathcal{T},\mathcal{L})$ with the resulting hiproof s.' The evaluation rules for a big step semantics for the language are given in Figure 4.3. Objects of the grammar type *prf* are operated on individually, whilst statements are operated on as a list (designated by the meta-variable *stmts*). Highlighting is used to make it clear the statement (and, for uniformity, *prf*) being operated on directly and :: represents the list 'cons' constructor. As with tactic evaluation the proof environment is contracted, writing $\mathcal{E}$ for $(\mathcal{T},\mathcal{L})$ when the environment is not modified. Recall that the tactic evaluation relation $\langle g, t \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g' \rangle$ is defined by the rules in Figure 3.23 on page 37.

These big step evaluation rules are intended to capture the meaning of the individual language elements directly. For completeness each rule is explained informally:

B-PRF-BLOCK A proof block operates on a single goal and consists of an *introduction tactic* t and a list of proof statements *stmts*. The introduction tactic is executed first, generating a hiproof s_1 and subgoals g. The subgoals are then passed to the statement list, which must evaluate successfully with another hiproof s_2. Thus, proof blocks are simply a specialised form of sequential composition: the resulting hiproof is s_1 ; s_2.

$$\frac{\langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, g\rangle \qquad \langle g, stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}{\langle [\gamma], \textsf{proof}(t)\ stmts\ \textsf{qed}\rangle \Downarrow_{\mathcal{E}} \langle s_1\ ;\ s_2\rangle} \quad \text{(B-Prf-Block)}$$

$$\langle [\gamma], \textsf{gap}\rangle \Downarrow_{\mathcal{E}} \langle id\rangle \quad \text{(B-Prf-Gap)}$$

$$\frac{\langle [\gamma], prf\rangle \Downarrow_{\mathcal{E}} \langle s\rangle}{\langle [\gamma], [l]\ prf\rangle \Downarrow_{\mathcal{E}} \langle [l]\ s\rangle} \quad \text{(B-Prf-Lab)}$$

$$\frac{\langle g_1, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, g_2\rangle \qquad \langle g_2, stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}{\langle g_1, \textsf{apply}\ t\ stmts\rangle \Downarrow_{\mathcal{E}} \langle s_1\ ;\ s_2\rangle} \quad \text{(B-Prf-App)}$$

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

$$\frac{\langle [\gamma], prf\rangle \Downarrow_{\mathcal{E}} \langle s_1\rangle \qquad \langle g, stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}{\langle \gamma :: g, \textsf{show}\ \gamma\ prf\ stmts\rangle \Downarrow_{\mathcal{E}} \langle ([show]\ s_1)\ \otimes\ s_2\rangle} \quad \text{(B-Prf-Show-2)}$$

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

$$\frac{\mathcal{L}(lem_1) = (\gamma_1, s_1) \quad \ldots \quad \mathcal{L}(lem_n) = (\gamma_n, s_n) \qquad \langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad \langle g, stmts\rangle \Downarrow_{\mathcal{E}} \langle s'\rangle}{\langle \gamma :: g, \textsf{from}\ lem_1\ \ldots\ lem_n\ \textsf{show}\ name\text{: }\gamma\ \textsf{by}\ t\ stmts\rangle \Downarrow_{\mathcal{E}} \langle ([name]\ s\ ;(([lem_1]\ s_1) \otimes \ldots \otimes ([lem_n]\ s_n))) \otimes s'\rangle} \quad \text{(B-Prf-From1)}$$

$$\frac{\mathcal{L}(lem_1) = (\gamma_1, s_1) \quad \ldots \quad \mathcal{L}(lem_n) = (\gamma_n, s_n) \qquad \langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad name \notin \mathcal{T} \wedge name \notin \mathcal{L}(\mathcal{L}) \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[name \longmapsto (\gamma,\ s\ ;(s_1 \otimes \ldots \otimes s_n))])} \langle s\rangle}{\langle g, \textsf{from}\ lem_1\ \ldots\ lem_n\ \textsf{have}\ name\text{: }\gamma\ \textsf{by}\ t\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle} \quad \text{(B-Prf-From2)}$$

$$\frac{variables(t) \subseteq \overline{X} \qquad name \notin \mathcal{T} \wedge name \notin \mathcal{L} \qquad (\mathcal{T},\mathcal{L}) \vdash \mathrm{t} \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T}[name \longmapsto (\overline{X},\ t)],\mathcal{L})} \langle s\rangle}{\langle g, \textsf{tac}\ name(\overline{X})\ :=\ t\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle} \quad \text{(B-Prf-Tac)}$$

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

Figure 4.3: Big step evaluation rules for Hiscript

B-PRF-GAP The **gap** command allows the omission of a proof for a particular goal. This is useful during the proof exploration process, when proof development may not necessarily be a linear process. The semantics captures gaps by returning the identity hiproof. This *passes* the unsolved goal through the rest of the hiproof, guaranteeing that it will still be valid.

B-PRF-LAB To evaluate a labelled proof step, the semantics first evaluates the proof step, then enclose the resulting hiproof in a labelled box.

B-PRF-APP Procedural proof steps — that is, raw tactic applications — are available through the **apply** command. The **apply** command is special as it is the only statement to operate on the full goal list. It applies the supplied tactic by appealing to the tactic evaluation relation and passes the resulting subgoals to the remaining statements. Again this is just a special case of sequential composition of the constructed hiproofs.

B-PRF-SHOW The **show** statement is the basic backwards proof command. It operates on the first goal in the proof context (if they match). The goal must be solved by the supplied *prf*. The rest of the goals are solved by the remaining statement. This time though, the hiproofs are composed in parallel using the tensor since it is solving individual and independent subgoals. The name of the **show** statement is utilised to introduce hierarchy in the underlying hiproof. Since names are optional, there is a version of this rule (B-PRF-SHOW-2) where no name is supplied. The only difference is that the hierarchy introduced is $[\mathit{show}]\ s$ instead of $[\mathit{show}\ n]\ s$, where n is the name given.

B-PRF-HAVE Local lemmas can be introduced using the **have** statement. The rule first checks that the name for the new lemma is *fresh*, then that the supplied proof is a good one. If so, the rest of the statements in the proof block can use the lemma by extending the environment to include the new lemma. Note that this rule limits the scope of the local lemma to be the rest of the statements in that proof block (and any nested proof blocks) but it will *not* be visible in parent proof blocks. The name freshness condition means that Hiscript has no local name overriding. The syntax for extending the environment is $\mathcal{L}[\mathit{name} \mapsto (\gamma, s)]$. This says the environment is extended so that *name* maps to a pair of goal γ and proof s. The correctness property for Hiscript (proved below) ensures that when the lemma environment is extended in this way, well-formedness is preserved.

B-PRF-FROM1 Given a list of lemmas with names $n_1, \ldots, n_n$, they can be composed in a forward manner to solve a goal γ using the **from** statement. By providing an appropriate justification t, the resulting hiproof can be composed in a backwards fashion.

B-PRF-FROM2 Similarly, it is possible to make forward steps that introduce a local lemma using this variant of the **from** statement.

B-PRF-TAC In analogy with local lemmas, Hiscript allows the introduction of local tactics using the **tac** statement. The rule checks well-formedness of the supplied tactic: appropriate tactic variables, fresh name, and well-formed.

```
show imptrans: (P → Q) ∧ (Q → R) → P → R
proof ( [intros] impl ; impl ; conjE)
  show R: P → Q, Q → R, P ⊢ R
  proof(mp)
    show P → Q, Q → R, P ⊢ Q → R by ax
    show Q: P → Q, P ⊢ Q
    proof(mp)
      show P → Q, P ⊢ P → Q gap
      show P → Q, P ⊢ P by ax
    qed
  qed
qed
```

Listing 4.5: A proof of $(P \to Q) \wedge (Q \to R) \to (P \to R)$

B-PRF-EMPTY Finally, when all statements have been executed (an empty statement list), there had better be no goals waiting to be solved.

The illustrate the structure of the hiproofs constructed from the evaluation rules, consider the proof in Listing 4.5, which uses the **gap** command in the proof of $P \to Q$ even though an appeal to the *ax* tactic would suffice. The hiproof of this script is shown in Figure 4.4; the trailing arrow represents the goal left unsolved by the use of the **gap** command.

The semantics always generates valid proofs:

Theorem 9 (Soundness of Hiscript big step semantics)**.** *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*.*

Proof. The proof is an induction on the height of the derivations and utilises the corresponding soundness property for hitac evaluation, Theorem 6 where appropriate.

For B-PRF-BLOCK, we know that $s_1 \vdash [\gamma] \longrightarrow g$ by Theorem 6 and that $s_2 \vdash g \longrightarrow g'$ for some g' by our induction hypothesis, thus by appealing to the validation rule V-SEQ it can be shown that $s_1 \ ; \ s_2 \vdash [\gamma] \longrightarrow g'$. A similar argument suffices for B-PRF-LAB and B-PRF-APP. The rule for an empty statement list, B-PRF-EMPTY, matches directly with V-EMPTY. A gap is validated with V-ID and (where $g = [\gamma]$). The rules B-PRF-HAVE, B-PRF-FROM2, and B-PRF-TAC simply require the induction hypothesis. For the rule B-PRF-FROM1, we know that each n_i maps to a hiproof s_i such that $s_i \vdash [\gamma_i] \longrightarrow g_i$ for some g_i. By Theorem 6 we also know that $s \vdash [\gamma] \longrightarrow [\gamma_1, \ldots, \gamma_n]$. Finally, by an appeal to the induction hypothesis for the rest of the statements we know $s' \vdash g \longrightarrow g'$ for some g'. Thus, put it all together using V-SEQ and V-TENS to show that the resulting hiproof is valid:

$$(s \ ; (s_1 \otimes \ldots \otimes s_n)) \otimes s' \vdash [\gamma] \longrightarrow g'.$$

Finally, we need to show soundness of the **show** statement rule, B-PRF-SHOW. This is also straightforward: two appeals to the induction hypothesis and the rule V-TENS gives the result. □

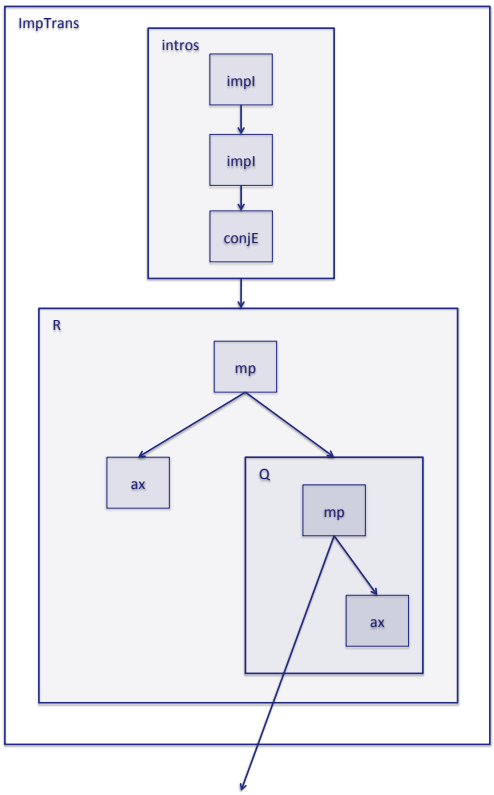

Figure 4.4: The hiproof constructed by evaluating Listing 4.5

The above proof shows that the rules are sound, completeness follows in a trivial way:

Theorem 10 (Completeness of big step semantics)**.** *If* $s \vdash \gamma \longrightarrow []$ *for a given environment* $(\mathcal{T}, \mathcal{L})$ *then there exists a prf such that* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$.

Proof. If s is a hiproof such that $s \vdash \gamma \longrightarrow []$ then, trivially, '**by** s' works when considering the hiproof as a hitac tactic. □

4.4 GAP-FREE PROOFS

Having gaps in a proof is cheating. Inspect the proof object — the hiproof — and the fraud can be spotted immediately. This is because it will have dangling goals, emerging from (potentially deep in the proof) where the **gap** command has inserted an identity. Figure 4.4 is a good example of this. To be more precise:

Definition 10 (Gap-free proof and statement)**.** A proof *prf* is gap-free if it is not a **gap** and, if it is a **proof** ... **qed** block then the contained statements are gap-free. A statement *stmt* is gap-free if it is an **apply**, **tac**, or **from** statement; or, if it is a **show** or **have** then the *prf* of that goal or lemma is gap-free.

The semantics ensures that gap-free proofs will generate hiproofs without any *danglers*.

Theorem 11 (Gap-free soundness)**.** *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$ *and prf is* gap-free *then* $s \vdash \gamma \longrightarrow []$.

Proof. Now, we know from Theorem 9 that $s \vdash \gamma \longrightarrow g$ for some g. Informally, note that out of all the evaluation rules, the only rule to introduce a discrepancy between the input goal list and the goal list evaluated by any hitac in the evaluations is the **gap** rule. Thus, if there are no gaps, then if a *prf* solves all its goals, so will the generated hiproof. □

This means that if there are gaps, then the list g contained exactly the gapped goals. This information can be useful for guiding the user in 'plugging' the gaps in their proof.

4.5 STATIC CHECKS ON PROOFS AND MINIMAL ENVIRONMENTS

The concepts of well-formedness and minimal environments extend naturally to proofs in Hiscript. In this section I formalise both notions. Well-formedness checking can detect potential evaluation failures without having to fully evaluate the proof: a simple form of type-checking. Minimal environments, on the other hand, provide a precise notion of the dependencies of a Hiscript proof. Well-formedness checking and minimal environments also play an important role in formalisation of refactorings in Part 2.

4.5.1 *Well-formedness checking*

Given a proof environment $(\mathcal{T}, \mathcal{L})$, the judgement:

$$(\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}$$

states that a proof *prf* is well-formed w.r.t. this environment. The notion of well-formed for proofs means all tactics used in the proof are well-formed and also that names introduced into the environment (by **have** statements, for example) are fresh. Since statements can introduce additional tactics and lemmas, the well-formedess judgement for statements needs to produce an updated environment:

$$(\mathcal{T}, \mathcal{L}) \vdash \mathit{stmt} : (\mathcal{T}', \mathcal{L}')$$

to say that the statement is well-formed in $(\mathcal{T}, \mathcal{L})$ and updates the environment to $(\mathcal{T}', \mathcal{L}')$. Figure 4.5 details the set of rules that inductively define the well-formedness judgement. The rules are straightforward and appeal to the appropriate tactic checking rules where necessary. Since proofs are not evaluated during the checking process, a 'fake' proof is added to the environment in the rule T-PRF-HAVE. It is straightforward to show:

Theorem 12 (Proofs that evaluate are well-formed)**.** *For a proof prf and a goal γ, if $\langle \gamma, \mathit{prf} \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$ then $(\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}$.*

Proof. The proof is an induction on the evaluation relation. □

$$\frac{(\mathcal{T},\mathcal{L}) \vdash t \qquad (\mathcal{T},\mathcal{L}) \vdash \mathit{stmts}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{proof}(t)\ \mathit{stmts}\ \textsf{qed}} \quad \text{(T-Prf-Block)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash \mathit{stmt} : (\mathcal{T}',\mathcal{L}') \qquad (\mathcal{T}',\mathcal{L}') \vdash \mathit{stmts}}{(\mathcal{T},\mathcal{L}) \vdash \mathit{stmt}\ \mathit{stmts}} \quad \text{(T-Prf-Stmts)}$$

$$(\mathcal{T},\mathcal{L}) \vdash \textsf{gap} \quad \text{(T-Prf-Gap)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash \mathit{prf}}{(\mathcal{T},\mathcal{L}) \vdash [l]\ \mathit{prf}} \quad \text{(T-Prf-Lab)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash t}{(\mathcal{T},\mathcal{L}) \vdash \textsf{apply}\ t : (\mathcal{T},\mathcal{L})} \quad \text{(T-Prf-App)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash \mathit{prf}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{show}\ \mathit{name} :\ \gamma\ \mathit{prf} : (\mathcal{T},\mathcal{L})} \quad \text{(T-Prf-Show)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash \mathit{prf}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{show}\ \gamma\ \mathit{prf} : (\mathcal{T},\mathcal{L})} \quad \text{(T-Prf-Show-2)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash \mathit{prf} \qquad \mathit{name} \notin \mathcal{T} \wedge \mathit{name} \notin \mathcal{L}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{have}\ \mathit{name} :\ \gamma\ \mathit{prf} : (\mathcal{T},\mathcal{L}[\mathit{name} \mapsto (\gamma, \mathit{id})])} \quad \text{(T-Prf-Have)}$$

$$\frac{(\mathcal{T},\mathcal{L}) \vdash t \qquad n_1 \in \mathcal{L} \ \ldots\ n_n \in \mathcal{L}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{from}\ n_1\ \ldots\ n_n\ \textsf{show}\ \mathit{name} :\ \gamma\ \textsf{by}\ t : (\mathcal{T},\mathcal{L})} \quad \text{(T-Prf-From1)}$$

$$\frac{\begin{array}{c}(\mathcal{T},\mathcal{L}) \vdash t \\ n_1 \in \mathcal{L} \ \ldots\ n_n \in \mathcal{L} \qquad \mathit{name} \notin \mathcal{T} \wedge \mathit{name} \notin \mathcal{L}\end{array}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{from}\ n_1\ \ldots\ n_n\ \textsf{have}\ \mathit{name} :\ \gamma\ \textsf{by}\ t : (\mathcal{T},\mathcal{L}[\mathit{name} \mapsto (\gamma, \mathit{id})])} \quad \text{(T-Prf-From2)}$$

$$\frac{\begin{array}{c}(\mathcal{T},\mathcal{L}) \vdash t \\ \mathit{variables}(t) \subseteq \overline{X} \qquad \mathit{name} \notin \mathcal{T} \wedge \mathit{name} \notin \mathcal{L}\end{array}}{(\mathcal{T},\mathcal{L}) \vdash \textsf{tac}\ \mathit{name}(\overline{X})\ :=\ t : (\mathcal{T}[\mathit{name} \mapsto (\overline{X}, t)], \mathcal{L})} \quad \text{(T-Prf-Tac)}$$

Figure 4.5: Hiscript well-formedness rules

4.5.2 *Minimal environments*

A minimal environment for proof evaluation is defined similarly to Hitac evaluation (as defined in Section 3.3.4). The slight subtlety in minimal environments for a proof *prf* is that statements in a proof can extend the environment and these *local* definitions should not be considered a part of the minimal environment. There is a definition for a *prf* and for a list of statements *stmts* because it is often useful to know the minimal environment for the remaining list of statements in a block:

Definition 11 (Minimal environment (proof and statements))**.** $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$ is a minimal environment for a proof, *prf* (respectively, list of statements, *stmts*), if:

1. $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}}) \vdash prf\ (stmts)$;
2. For every proof environment $(\mathcal{T}', \mathcal{L}')$ such that $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$, the proof (statements) are not well-formed.

Given an environment $(\mathcal{T}, \mathcal{L})$ and a proof *prf* that is well-formed under that environment, define the environment $(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})$ as follows:

$$\begin{aligned} \mathcal{T}|_{prf} &= \{\ (n, \mathcal{T}(n))\ \mid\ n \in (tacs(prf) \setminus localnames(prf))\ \} \\ \mathcal{L}|_{prf} &= \{\ (n, \mathcal{L}(n))\ \mid\ n \in (lemmas(prf)\ \setminus localnames(prf))\ \} \end{aligned}$$

where *tacs* and *lemmas* are extensions to those defined inductively on the structure of hitac tactics in Section 3.3.4. The function *localnames* returns all the local definitions (tactics and lemmas) in a proof block. As expected, this turns out to be a minimal environment:

Theorem 13 (Minimality of $(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})$)**.** *Given an initial environment* $(\mathcal{T}, \mathcal{L})$ *and a proof* *prf*, *the environment* $(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})$, *defined above, is a minimal environment.*

Proof. The proof is similar to the hitac equivalent, except caution is needed when dealing with local definitions. □

A similar definition and theorem applies for statements and is detailed in Section A.1.1 of Appendix A alongside some other basic properties about minimal environments.

4.5.3 *Environment extension*

The general property of closure under environment extension proved in Theorem 8 does not extend to Hiscript proofs. To see why, consider extending a tactic environment $\mathcal{T}$:

$$\mathcal{T}' := \mathcal{T}[newtac \mapsto ([], id)]$$

Now imagine a proof, well-formed under $(\mathcal{T}, \mathcal{L})$, of the form shown in Listing 4.6. It will fail well-formedness checking under the extended environment $(\mathcal{T}', \mathcal{L})$ since the name *newtac* will already exist in the environment: a failure of the precondition to the rule T-Prf-Tac. Thus, the extended environment must be restricted to exclude locally defined names.

```
proof
 ...

tac newtac := ...

 ...
qed
```

Listing 4.6: A proof introducing a local tactic definition *newtac*

Theorem 14 (Closure of *prf* well-formedness under environment extension)**.** *Assume* $(\mathcal{T}, \mathcal{L}) \subset (\mathcal{T}', \mathcal{L}')$ *and* $(\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}$*. If*

$$\mathit{localnames}(\mathit{prf}) \cap (\mathit{names}(\mathcal{T}') \cup \mathit{names}(\mathcal{L}')) = \{\}$$

then $(\mathcal{T}', \mathcal{L}') \vdash \mathit{prf}$*, where* ***localnames*** *is defined inductively on prf.*

Proof. By induction on the proof well-formedness judgement with appropriate calls to the equivalent theorem for tactic well-formedness, Theorem 8. □

A similar results holds for a list of statements and is given in Section A.1.2 of Appendix A.1.2 alongside other properties of environment extension.

4.6 DISCUSSION

This chapter introduced Hiscript and demonstrated some theoretical properties:

- The language is sound. Successful evaluation constructs valid hiproofs.
- Minimal dependencies of a proof can be calculated.
- A notion of well-formedness can statically tell if certain types of errors are present.

Hiscript also has the novel feature of constructing hierarchical proofs, which could be utilised in a *proof viewer* tool to allow one to focus in on important parts of the proof or even just view the proof at its high-level detail. Interestingly, the proof language itself constructs hiproofs with a rigidly defined structure that is close to the notion of TNF from Chapter 3. In particular, a proof block:

```
proof(t)
  show γ1 prf
  ...
  show γn prf
qed
```

will construct hiproofs of the form $s \;;\; s_1 \otimes \ldots \otimes s_n$ where s is the hiproof constructed by evaluating t and each s_i is the proof of γ_i. This is a slightly weaker version of TNF, but this actually gives a direct translation of any hiproof into Hiscript. I do not give any details of this translation here; rather, I provide it (and a more powerful translation) in Chapter 9. Note that this is different to the trivial characterisation given by Theorem 10 on page 53.

4.6.1 *Comparison with other declarative languages*

Hiscript is a generic declarative proof language, so it loses the logic-specific constructs that are available for a proof language developed specifically for one system. The `miz3` language is a good example, with commands like `assume` corresponding to implication introduction (Wiedijk, 2012). This is also true for Isar — the language most closely related to Hiscript — although by the nature of Isabelle as a logical framework, Isar has proof commands which directly correspond to the Isabelle/Pure logic. One of the deficiencies of Hiscript, stemming from its genericity, is the lack of constructs for manipulating assumptions and variables in the proof. For example, in Isabelle a goal can be stated as:

```
fix x y : nat
assume *: x < y
show x+2 < y+2
```

which, depending on the goal representation, in Hiscript would have to look something like:

```
show x : nat, y :nat, x < y ⊢ x+2 < y+2
```

which is less readable. Hiscript could be extended with these constructs if it assumed more structure to the notion of *derivation system* that underlies hiproofs; furthermore, I could also allow the Hiscript language to be extended, upon instantiation, by additional constructs that map directly to atomic tactics in the hiproof framework.

When compared to Isar, Hiscript also lacks many of the convenient structuring mechanisms and syntactic abbreviations of a fully-fledged proof language Wenzel (1999). Furthermore, it lacks sophisticated language constructs for dealing directly with common proof techniques such as induction, case analysis and equational reasoning. These constructs allow elegant and readable proofs to be written in Isar. Isar is also a *generic* language. It is parameterised by a justification language known as *methods* and many of the constructs are logic independent. In particular, it can be used uniformly for any of the object logics in Isabelle, not just Isabelle/HOL. However, in practice, the language is intimately tied to the underlying meta-logic of Isabelle: Isabelle/Pure.

In Isabelle, Isar methods are based on the underlying tactic language. This makes it more challenging to reason precisely about dependencies of a particular lemma, since the most common tactics in Isabelle: *simp* and *auto* are sophisticated search procedures.

5

PROOF DOCUMENTS

5.1 INTRODUCTION

In the previous two chapters, I have shown how to:

- Check simple properties of Hitac and Hiscript using a well-formedness judgement.
- Evaluate Hiscript proofs and tactics against a given goal (list) to construct a proof object, a hiproof.

Both of these properties are defined with respect to a *proof environment* $(\mathcal{T}, \mathcal{L})$ as defined in Section 3.3 on page 36.

I define proof environments as the semantic object constructed by sets of lemmas and tactic definitions. Simply put: *formal theories build proof environments*. What stuff goes in theories? Isabelle has quite a complicated theory infrastructure with dedicated *theory* files in which axioms can be stated, lemmas can be proved, definitions can be made, tactics defined and notation introduced (Nipkow et al., 2002). On the other side of the spectrum, the HOL Light system does not have a separate notion of proof document: there are simply files in the underlying programming language, OCaml (Harrison, 1998). Hiscript sits somewhere in the middle: I provide a dedicated language for constructing proof documents — I will also call it Hiscript — but for simplicity I currently only allow lemmas and tactics in the language.

CHAPTER MAP This chapter introduces a language for constructing *theories*, where the underlying tactic language is Hitac and proofs can be constructed using Hiscript. I give a formal semantics for theories, based on the linear ML-style model of evaluation, where evaluating a theory will construct a proof environment. Since the primary purpose is to investigate semantics for proof languages and, in particular, from a generic standpoint, theories consist only of tactic definitions and lemmas.

I start with a model of interaction for single theory files in Section 5.2 then, Section 5.3 postulates a model for sets of theories with a simple import structure. A collection of theories is called a *proof document*. Finally, Section 5.4, briefly summarises this chapter.

CONTRIBUTIONS This chapter follows from the investigation of proof language semantics with a study of *theory* semantics. Specifically, this chapter contains the following contributions:

1. A formal semantics for theories that separates static *well-formedness* checks from *proof checking*.
2. A formal semantics for a simple theory import mechanism for theories that also includes an export interface mechanism using the *public/private* notion known from object-oriented programming.
3. The semantics are proved to construct *well-formed* proof environments.

The single theory part of this work, described in Section 5.2 has been published in Whiteside et al. (2011).

5.2 THEORIES AND THEIR SEMANTICS

In this section, I look at the simplest type of theory: a single collection of all the tactics and lemmas used in a proof document.

5.2.1 *Theory syntax*

A theory is viewed abstractly as a (named) list of lemmas and tactic definitions. The grammar is given in Figure 5.1. Theories consist of a named sequence of lemmas and

Hiscript theory syntax

$$
\begin{array}{lcl}
\textit{theory} & ::= & \textbf{theory}\ \textit{name} \\
 & & \quad \textit{thyitem}^{*} \\
 & & \textbf{end} \\
 & & \\
\textit{thyitem} & ::= & \textbf{begin} \\
 & | & \textbf{tac}\ \textit{name}(X_1, \ldots, X_n) := t \\
 & | & \textbf{hitac}\ \textit{name}(X_1, \ldots, X_n) := t \\
 & | & \textbf{lemma}\ \textit{name}\textbf{:}\ \textit{goal} \\
 & & \quad \textit{prf}
\end{array}
$$

Figure 5.1: Hiscript theory syntax

tactics, enclosed within **begin** and **end** tags. Elements of this sequence are called *theory items*. A **lemma** can be seen as a *global* form of a **have** statement in Hiscript. Lemmas consist of a name, a goal, and a formal proof *prf* of that goal in Hiscript. There are two types of tactic definition: **tac** will simply store and execute the supplied tactic;

and **hitac** will implicitly add an additional hierarchical label to the supplied tactic. Thus:

```
hitac mytac := t
```

is equivalent to:

```
tac mytac := [mytac] t
```

The reason for separate constructs is because it is informative to label the result of tactic execution; however, this should not be a default behaviour as any tactic definition that can be evaluated on a list of $n > 1$ goals will fail to evaluate when labelled, because a hierarchical box must have only one input. If $t \equiv t_1 \otimes t_2$, for example, then $[l]\ t$ would not construct a valid hiproof. Tactic definitions can also be parameterised (also known as *tacticals*), X_i are tactic variables that can occur within t and must be instantiated when the tactic is used.

5.2.2 *Semantics for theories*

The key idea behind theory evaluation is the stepwise extension of the proof environment. The first theory item in a theory must evaluate with an empty environment. This item is then added to the environment and can then be used by later theory items. Thus, theory evaluation must always construct *well-formed* environments. Following the approach for Hiscript proofs, simple structural checks are given by a judgement:

$$\vdash \mathit{theory} : \langle \mathcal{T}, \mathcal{L} \rangle$$

which performs some basic well-formedness checks on the structure of the theory and separate the mundane, static checks from the more exciting proof-checking process. Well-formedness ensures, for example, that each lemma or tactic has a unique name and that, with respect to the environment constructed thus far, each tactic and lemma is well-formed (using the judgements $(\mathcal{T}, \mathcal{L}) \vdash t$ and $(\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}$ respectively).

Full-blown evaluation — that is, evaluation of a proof — is only possible for well-formed proof documents and is given by a judgement

$$\vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

The rules in Figure 5.2 inductively define the well-formedness checks. A proof document itself is well-formed if the constituent theory items are well-formed. The only theory item that is well-formed for an empty environment is a **begin** item, which ensures this is the first item in a proof document. Tactic definitions and lemmas will extend the environment if they are well-formed in the environment constructed thus far. Tactics are well-formed if the name is fresh and the tactic is well-formed as per the definition in Section 3.3.3. Checking a lemma amounts to checking that the name is fresh and that the proof is well-formed as per the definition in Section 4.5.1. It is important to mention again that there is no explicit evaluation with the lemmas. Thus, it is possible that one or all of the lemmas will not evaluate to a successful hiproof. While constructing an environment plays an important role in well-formedness checking and (as shall be seen later) is actually a well-formed environment, it's a fake: there are no proofs, just the equivalent of gaps.

$$\frac{\vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \textsf{theory}\ \mathit{name}\ \mathit{thyitems}\ \textsf{end} : \langle \mathcal{T}, \mathcal{L} \rangle}\ \text{(T-Thy)}$$

$$\vdash \textsf{begin} : \langle \{\}, \{\} \rangle\ \text{(T-Begin)}$$

$$\frac{\vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathfrak{n} \notin \mathit{names}(\mathit{thyitems}) \qquad \mathit{vars}(t) \subseteq \overline{X} \qquad (\mathcal{T}, \mathcal{L}) \vdash t}{\vdash \mathit{thyitems}\ \textsf{tac}\ \mathfrak{n}(\overline{X}) := t : \langle \mathcal{T}[\mathfrak{n} \mapsto (\overline{X}, t)], \mathcal{L} \rangle}\ \text{(T-Tac)}$$

$$\frac{\vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathfrak{n} \notin \mathit{names}(\mathit{thyitems}) \qquad \mathit{vars}(t) \subseteq \overline{X} \qquad (\mathcal{T}, \mathcal{L}) \vdash t}{\vdash \mathit{thyitems}\ \textsf{hitac}\ \mathfrak{n}(\overline{X}) := t : \langle \mathcal{T}[\mathfrak{n} \mapsto (\overline{X}, [n]\ t)], \mathcal{L} \rangle}\ \text{(T-Hitac)}$$

$$\frac{\vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathfrak{n} \notin \mathit{names}(\mathit{thyitems}) \qquad (\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}}{\vdash \mathit{thyitems}\ \textsf{lemma}\ \mathfrak{n} \colon \gamma\ \mathit{prf} : \langle \mathcal{T}, \mathcal{L}[\mathfrak{n} \mapsto (\gamma, \mathit{id})] \rangle}\ \text{(T-Lemma)}$$

Figure 5.2: Checking well-formedness of theories

It is possible for a proof document to be well-formed, but not evaluate successfully. Consider the document given in Listing 5.1, for example. Both the tactic and procedural proof are well-formed but the *intros* tactic defined in the document does not perform conjunction introduction; thus, the proof will fail to evaluate.

To properly check the lemmas, the proof document must be evaluated. Evaluation is given by the rules in Figure 5.3. A proof document evaluates successfully to an envi-

$$\frac{\vdash \textsf{theory}\ \mathit{name}\ \mathit{prf}\ \textsf{end} : \langle \mathcal{T}', \mathcal{L}' \rangle \qquad \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \textsf{theory}\ \mathit{name}\ \mathit{prf}\ \textsf{end} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}\ \text{(Thy-Eval)}$$

$$\vdash \textsf{begin} \Downarrow \langle \{\}, \{\} \rangle\ \text{(Begin-Eval)}$$

$$\frac{\vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \mathit{thyitems}\ \textsf{tac}\ \mathfrak{n}(\overline{X}) := t \Downarrow \langle \mathcal{T}[\mathfrak{n} \mapsto (\overline{X}, t)], \mathcal{L} \rangle}\ \text{(Tac-Eval)}$$

$$\frac{\vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \mathit{thyitems}\ \textsf{hitac}\ \mathfrak{n}(\overline{X}) := t \Downarrow \langle \mathcal{T}[\mathfrak{n} \mapsto (\overline{X}, [n]\ t)], \mathcal{L} \rangle}\ \text{(Hitac-Eval)}$$

$$\frac{\vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad \langle [\gamma], \mathit{prf} \rangle \Downarrow_{(\mathcal{T}, \mathcal{L})} \langle s \rangle}{\vdash \mathit{thyitems}\ \textsf{lemma}\ \mathfrak{n} \colon \gamma\ \mathit{prf} \Downarrow \langle \mathcal{T}, \mathcal{L}[\mathfrak{n} \mapsto (\gamma, [n]\ s)] \rangle}\ \text{(Lemma-Eval)}$$

Figure 5.3: Evaluation rules for theories

ronment $(\mathcal{T}, \mathcal{L})$ if it is well-formed (the environment constructed by well-formedness checking can be ignored) and if the theory items evaluate to construct the environment $(\mathcal{T}, \mathcal{L})$.

Tactics are now added to the environment without question — since by well-formedness checking they are suitable — with the rules Tac-Eval and Hitac-Eval

```
theory wfnoteval
begin

hitac intros := impl ; ALL(intros) | ID

lemma noconj (P ⇒ P) ∧ (Q ⇒ Q)
apply intros ; ax ⊗ ax
end
```

Listing 5.1: A well-formed proof document that does not evaluate

and the rule LEMMA-EVAL uses the Hiscript evaluation relation with the proof environment constructed thus far:

$$\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle.$$

5.2.3 *Correctness of evaluation*

The key correctness property during evaluation of a proof document is the assurance that any proof environment constructed by a theory must be well-formed. For a proof environment to be well-formed, the individual tactic and lemma environments need to be well-formed. For a lemma environment — storing goals and hiproofs — all the associated goals must be validated by the associated hiproofs. That is:

$$\text{for every } name, \text{ if } \mathcal{L}(name) = (\gamma, s) \text{ then } s \vdash \gamma \longrightarrow g \text{ for some } g.$$

Tactic environments, however, are well-formed if they are closed under names. That is, any referenced tactic or lemma name in the definition of another tactic also exists in the environment. This property is defined by a judgement

$$\mathcal{L} \vdash \mathcal{T}$$

as defined by the rules in Figure 3.25. Thus, the appropriate correctness property states that 'if a proof document evaluates successfully, then the constructed environment is well-formed'. That is:

Theorem 15 (Correctness of proof document evaluation)**.** *If*

$$\vdash theory \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

then:

- *for every name, if* $\mathcal{L}(name) = (\gamma, s)$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*;*
- $\mathcal{L} \vdash \mathcal{T}$.

Proof. First, note that the environment constructed by the well-formedness judgement is also well-formed:

Lemma 16 (Correctness of proof document well-formedness)**.** *If*

$$\vdash \textit{theory} : \langle \mathcal{T}, \mathcal{L} \rangle$$

then:

- *for every name, if* $\mathcal{L}(name) = (\gamma, s)$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*;*
- $\mathcal{L} \vdash \mathcal{T}$.

Proof of lemma. Now, every element of $\mathcal{L}$ has an identity proof therefore if $\mathcal{L}(name) = (\gamma, s)$ then $s \equiv id \vdash \gamma \longrightarrow \gamma$, thus, the lemma environment is well-formed. The proof that $\mathcal{L} \vdash \mathcal{T}$ is a simple induction on the well-formedness rules for proof documents, referencing the rules for well-formedness checking of tactic environments, given in Figure 3.25. For the base case (the rule T-BEGIN) the tactic environment is empty, which can be matched with TE-EMPTY directly. Then, for the inductive cases, the rules T-TAC, T-HITAC and, T-LEMMA match directly with the rule TE-CONS or the inductive hypothesis (for T-LEMMA, since the tactic environment is not changed). □

Using this lemma and the observation that well-formedness checking and evaluation of proof documents constructs the same tactic environment, we only need to show well-formedness of the lemma environment and this follows directly from soundness of the Hiscript evaluation relation (Theorem 9). □

As with individual Hiscript proofs, a stronger well-formedness condition on lemmas may be given:

$$\text{for every } \textit{name}, \text{ if } \mathcal{L}(name) = (\gamma, s) \text{ then } s \vdash \gamma \longrightarrow []$$

if all the proofs are gap-free.

5.2.4 *Example theory*

Example 3 (A first theory)**.** Figure 5.4 provides a simple example of a proof document that is both well-formed and evaluates successfully. Alongside, Figure 5.5 gives an indication of the state of the proof environment at different stages in the evaluation. For example, after evaluating Line 2, the environment is empty. Then, after evaluating up to and including Line 6, the tactic environment has been extended to contain the three tactic definitions. These definitions (and REP in particular) are used to successfully check well-formedness of Line 8, allowing the tactic environment to be further extended. After Line 8, the tactic environment now contains four items, but the lemma environment is still empty. The lemma environment is extended to contain lem_1 by the time the proof document is fully evaluated, at Line 16. The importance of well-formedness checking is seen before proof evaluation on Line 11 as evaluation of the tactic intros involves an unfolding to its definition, which uses REP. Furthermore, when REP is unfolded, it relies upon both ID and ALL. Thus, if any of these definitions are missing from the proof document, then proof evaluation would fail. The hiproof stored in $\mathcal{L}$ as the proof for lem_1 can be seen in Figure 5.6.

```
1  theory example
2  begin
3
4  tac ALL(X) := X ⊗ (ALL(X) | ⟨⟩ )
5  tac ID := ALL(id)
6  tac REP(X) := X ; ALL(REP(X)) | ID )
7
8  hitac  intros  := REP( impI | conjI  )
9
10 lemma lem_1: P ∧ Q ⇒ Q ∧ P
11 proof( intros )
12   show q: P ∧ Q ⊢ Q by conjE ; ax
13   show p: P ∧ Q ⊢ P by conjE ; ax
14 qed
15
16 end
```

Figure 5.4: A simple proof document

2	$\mathcal{T} = \{\}$ and $\mathcal{L} = \{\}$
6	$\mathcal{T} = \{ALL \mapsto \ldots, ID \mapsto \ldots, REP \mapsto \ldots\}$, $\mathcal{L} = \{\}$
8	$\mathcal{T} = \{\ldots, REP \mapsto \ldots, intros \mapsto \ldots\}$, $\mathcal{L} = \{\}$
16	$\mathcal{T} = \{\ldots, intros \mapsto \ldots\}$, $\mathcal{L} = \{lem_1 \mapsto \ldots\}$

Figure 5.5: Environment during evaluation

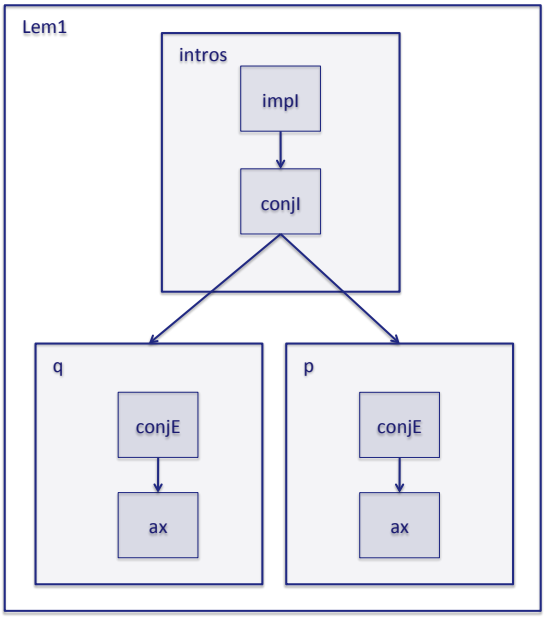

Figure 5.6: Hiproof constructed by the proof of lem_1

Example 4 (A larger theory about sets)**.** Figure 5.8 shows a larger theory containing a development in set theory, which uses the atomic rules shown in Figure 5.7. In this example, general tacticals and specific tactics are defined, then lemmas about the union and intersection operations are proved. Here, the potential advantages for a modular approach to theory files, the subject of the next section, can be seen for the first time. Lemmas are used frequently, like *emptysubany*.

$$\frac{\Gamma \vdash A \subseteq B \qquad \Gamma \vdash B \subseteq A}{\Gamma \vdash A = B} \quad (=\text{-INTRO})$$

$$\frac{\Gamma, A \subseteq B \vdash C \qquad \Gamma, B \subseteq A \vdash C}{\Gamma, A = B \vdash C} \quad (=\text{-ELIM})$$

$$\frac{\Gamma, x \in A \vdash x \in B}{\Gamma \vdash A \subseteq B} \quad (\subseteq\text{-INTRO})$$

$$\frac{\Gamma \vdash x \in A \qquad \Gamma \vdash x \in B}{\Gamma \vdash x \in A \cap B} \quad (\cap\text{-INTRO})$$

$$\frac{\Gamma, x \in A, x \in B \vdash C}{\Gamma, x \in A \cap B \vdash C} \quad (\cap\text{-ELIM})$$

$$\frac{\Gamma \vdash x \in A}{\Gamma \vdash x \in A \cup B} \quad (\cup\text{-INTRO1})$$

$$\frac{\Gamma \vdash x \in B}{\Gamma \vdash x \in A \cup B} \quad (\cup\text{-INTRO2})$$

$$\frac{\Gamma, x \in A \vdash C \qquad \Gamma, x \in B \vdash C}{\Gamma, x \in A \cup B \vdash C} \quad (\cup\text{-ELIM})$$

$$\frac{}{\Gamma, x \in \varnothing \vdash A} \quad (\text{EMPTY})$$

$$\frac{A \subseteq C \qquad B \subseteq C}{A \cup B \subseteq C} \quad (\cup\text{-SMALLEST})$$

Figure 5.7: Extended set of atomic tactics for the SET hiproof instantiation

5.3 PROOF DOCUMENTS

Most proof assistants allow the proof engineer to structure a development into multiple theories that are shorter and easier to understand than the whole, helping to keep the proof environment to a practical minimum size. The theory shown in Figure 5.8 could, for example, be split into a *basics* theory that contains all the tactic definitions and some simple lemmas. Then, the proofs about union and intersection could be

```
theory set
begin

tac ALL(X) := X ⊗ (ALL(X) | ⟨⟩ )
tac ID := ALL(id)
tac REPEAT(X) := X ; ALL(REPEAT(X)) | ID

tac intro := =−intro | ⊆−intro | ∩−intro
hitac intros := REPEAT(intro)

lemma intersubleft: A ∩ B ⊆ A
proof( intros )
show x ∈ A ∩ B ⊢ x ∈ A
  by ∩−elim ; ax
qed

lemma intersubright: A ∩ B ⊆ B
proof( intros )
show x ∈ A ∩ B ⊢ x ∈ B
  by ∩−elim ; ax
qed

lemma unsubleft: A ⊆ A ∪ B
  by ⊆−intro ; ∪−intro1 ; ax

lemma unsubright: B ⊆ A ∪ B
  by ⊆−intro ; ∪−intro2 ; ax

lemma emptysubany: ∅ ⊆ A
proof( intros )
  show x ∈ ∅ ⊢ x ∈ A
    by empty
qed

lemma interempty: A ∩ ∅ = ∅
proof( intros )
  show A ∩ ∅ ⊆ ∅
    by lem intersubright
  show ∅ ⊆ A ∩ ∅
    by lem emptysubany
qed

.
.
.
```

```
.
.
.

lemma unempty: A ∪ ∅ = A
proof( intros )
  show A ⊆ A ∪ ∅
    by lem unsubleft
  show A ∪ ∅ ⊆ A
  proof (∪−smallest)
    show A ⊆ A
      by ax
    show ∅ ⊆ A
      by emptysubany
  qed
qed

lemma AunA: A ∪ A = A
proof(=−intro)
  show AunAsubA: (A ∪ A) ⊆ A
  proof( intros )
    show x ∈ (A ∪ A) ⊢ x ∈ A
      by ∪−elim ; ALL(ax)
  qed
  show AsubAunA: A ⊆ (A ∪ A)
    by lem unsubleft
qed


lemma AintA: A ∩ A = A
proof(=−intro)
  show A ∩ A ⊆ A
    by lem intersubleft
  show A ⊆ A ∩ A
  proof (⊆−intro)
    show x ∈ A ⊢ x ∈ A ∩ A
      by ∩−intro ; ax ⊗ ax
  qed
qed

.
.
.

end
```

Figure 5.8: A fragment from a proof document with simple set theory lemmas

split into their own theory file, before being merged in a more general theory about sets. I call a collection of theories a *proof document*.

In this section, I study one of the simplest methods for creating modular structure in a proof document. It can best be described as a simple theory import mechanism. A proof document will then be a set of theories and the preamble for each theory will then look like:

```
theory mythy
import thyA
import thyB
begin
  ...
```

with the results from *thyA* and *thyB* immediately available in *mythy*, rather than starting from scratch. All practical proof assistants provide this style of theory.

Introducing a theory import mechanism as seen above raises some design questions:

- If one theory imports another, does it also import anything imported by the other? That is: is the notion of importing *transitive*?
- Should the language provide a mechanism for restricting which theory items are imported when that theory is imported by another? That is, does each theory define an *export interface*?
- Does each theory have a separate namespace or is there a global namespace? More precisely, this question asks how potential names clashes are always avoided, particularly when one can develop independent strands, that may join in the future.

For Hiscript, I answer yes to the first two questions. Importing theories is a transitive relation, with duplication avoided by using a set-theoretic union operation during the import process. Furthermore, I introduce **private** and **public** annotations (inspired by object-oriented programming languages) to each theory item to determine whether they should be visible when their local theory is imported. To handle namespaces, I take the approach that each theory has a local namespace. When a theory, named *thyA* say, is imported by another, its public items must be referred to by a *fully qualified name*: *thyA.item*. Thus, uniqueness of names is provided by a combination of uniqueness of names in their own theory and uniqueness of theory names. In what follows, when I refer to the single theory approach in Section 5.2, I call a theory in this style a *basic theory* to avoid confusion with a single theory, which is a member of a proof document.

The next section introduces the syntax for proof documents. Then, Section 5.3.2 introduces the semantics for evaluating a theory in a proof document and is proved correct in Section 5.3.2.1. Some simple properties of minimal environments and environment extension are introduced in Section 5.3.3 and Section 5.3.4 gives an evaluation relation for evaluating all theories in a proof document. Finally, Section 5.3.5 describes an example proof document.

5.3.1 *Proof document syntax*

Proof documents syntax

$$
\begin{array}{lcl}
\textit{document} & ::= & \textit{theory}^* \\
\\
\textit{theory} & ::= & \textsf{theory}\ \textit{name} \\
 & & \textit{import}^* \\
 & & \textsf{begin} \\
 & & \quad (\textit{visibility}\ \textit{thyitem})^* \\
 & & \textsf{end} \\
\\
\textit{visibility} & ::= & \textsf{public} \mid \textsf{private} \\
\\
\textit{import} & ::= & \textsf{import}\ \textit{name} \\
 & \mid & \textsf{import}\ \textit{name}_1\ \textsf{as}\ \textit{name}_2 \\
\\
\textit{thyitem} & ::= & \textsf{tac}\ \textit{name}(X_1, \ldots, X_n) := t \\
 & \mid & \textsf{hitac}\ \textit{name}(X_1, \ldots, X_n) := t \\
 & \mid & \textsf{lemma}\ \textit{name}\textsf{:}\ \textit{goal} \\
 & & \quad \textit{prf}
\end{array}
$$

Figure 5.9: Syntax for multiple theory files

The syntax for a basic theory is extended (and modified slightly) to represent proof documents as shown in Figure 5.9. A *document* is a list of theories and each individual theory can have zero or more *import* statements followed by zero or more theory items, each attached with a visibility. The visibility annotation is either **public** or **private** and is used to determine the items from a theory that are imported when a theory is included in the import list of another. Any imported theories can be imported with their original name or, using the **as** command, can be renamed. Public theory items that have been imported will then be referred to using their *fully qualified name*. Name qualification is handled by the 'dot' notation: the public lemma *lem1* from theory *A* will be referred to as *A.lem1* in descendent theories. A function *imports* returns the transitive closure of all import statements for a given theory. For a proof document *document*, the set of theories must be free from *cyclic dependencies*:

$$\forall\, thy \in document.\ thy \notin imports(thy)$$

5.3.2 *Evaluating theories in a document*

To evaluate a theory within a proof document, one must first evaluate all of the necessary imports. It is important to note, though, that this does not necessarily mean that evaluating every theory in the document. The general idea of evaluation is the same as for basic theories: splitting the well-formedness checks and proof evaluation into two separate steps. This approach is particularly useful as evaluating a theory with imports requires more sophisticated well-formedness checks than a basic theory. For example, private items must not be referenced in descendant theories and there must not be any name clashes in the import lists.

Since theory items now contain additional information (their visibility), the maps $\mathcal{T}$ and $\mathcal{L}$ are extended:

$$\mathcal{L} : name \to (visibility \times goal \times s)$$

$$\mathcal{T} : name \to (visibility \times var\ list \times t).$$

Note that this also requires adding a visibility to any local lemmas or local tactics introduced within declarative proofs. It is assumed that these are always introduced as **private** items. This allows me to define the projections

$$vis_{\mathcal{T}}(n) = fst(\mathcal{T}(n))$$

$$vis_{\mathcal{L}}(n) = fst(\mathcal{L}(n))$$

which can be used to introduce the notion of an *environment restriction* for a tactic environment:

$$\mathcal{T}^{pub} = \{(n, \mathcal{T}(n)) \mid \forall n \in \mathcal{T}.\ vis_{\mathcal{T}}(n) = \textbf{public}\}$$

and similarly for the lemma environment:

$$\mathcal{L}^{pub} = \{(n, \mathcal{L}(n)) \mid \forall n \in \mathcal{L}.\ vis_{\mathcal{L}}(n) = \textbf{public}\}.$$

These restrictions can ensure that only public items are used in any descendant proof document. This and other well-formedness checks are performed by the well-formedness check for a theory:

$$\mathcal{D} \vdash theory : \langle \mathcal{T}, \mathcal{L} \rangle$$

where $\mathcal{D}$ is a map $\mathcal{D} : name \to theory$, which translates a theory name to its body, providing a simple mechanism for referring to the theories in a document. The individual theory items are defined with the rule:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle$$

where $(\mathcal{T}_i, \mathcal{L}_i)$ is the imported environment. Both these relations are defined by the rules in Figure 5.10. I assume a substitution mechanism on theories: $theory[id := name.id]$ for qualifying all local names. That is, it would transform a theory:

```
theory B begin
import A
begin

public tac simpletac := ...  A.x  ...

public lemma lem_1: ...
  by simpletac

end
```

to

```
theory B begin
import A
begin

public tac B.simpletac := ...  A.x  ...

public lemma B.lem_1: ...
  by B.simpletac

end
```

Note that the substitution will not rename any previously qualified names: thus, A.x is not renamed to B.A.x.

There are two types of rules: *import* rules and *theory item* rules. The top-level theory check (T-THY) first checks all the imports before checking the sequence of theory items. The import rules build up the imported environment by checking each imported proof document individually then joining the imported environments. The rule T-IMPORT-EMPTY deals with the possibility of an empty import list. When performing an **import as**, the new name must not already be used by a theory in the document. Rather than using a basic union operation ($\cup$) to join together environments, I use a version that:

- uses the name as the means of deciding whether two items are identical;
- in the case of duplicates will prefer the item from the left environment.

This is known as the *map override* function on relations, represented with the symbol $\lhd$, thus I define:

$$\mathcal{T}_2 \lhd \mathcal{T}_1 = \mathcal{T}_1 \cup \{(n, \mathcal{T}_2(n)) \mid n \notin \mathcal{T}_1\}$$

that is: 'the map override between $\mathcal{T}_1$ and $\mathcal{T}_2$ consists of the environment $\mathcal{T}_1$ and those items in $\mathcal{T}_2$ whose names are not contained in $\mathcal{T}_1$.' Each imported proof document is transformed to use fully qualified names before checking, using the substitution $[id := A.id]$, which can be defined on the structure of theories. In practice, this means that any duplicates arising during imports comes from the same ancestor theory being imported by more than one ancestor of the current theory; thus, both the name and the body of the environment will be identical.

$$\frac{\mathcal{D} \vdash \mathit{imports} : \langle \mathcal{T}_i, \mathcal{L}_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle}{\mathcal{D} \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} : \langle \mathcal{T}_i \trianglelefteq \mathcal{T}, \mathcal{L}_i \trianglelefteq \mathcal{L} \rangle} \quad \text{(T-Thy)}$$

$$\mathcal{D} \vdash [] : \langle \{\}, \{\} \rangle \quad \text{(T-Import-Empty)}$$

$$\frac{\mathcal{D} \vdash \mathcal{D}(\mathrm{A})[\mathit{id} := \mathrm{A}.\mathit{id}] : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathcal{D} \vdash \mathit{imports} : \langle \mathcal{T}_i, \mathcal{L}_i \rangle}{\mathcal{D} \vdash \textbf{import}\ \mathrm{A}\ \mathit{imports} : \langle \mathcal{T}_i \trianglelefteq \mathcal{T}, \mathcal{L}_i \trianglelefteq \mathcal{L} \rangle} \quad \text{(T-Import-1)}$$

$$\frac{\mathrm{B} \notin \mathcal{D} \qquad \mathcal{D} \vdash \mathcal{D}(\mathrm{A})[\mathit{id} := \mathrm{B}.\mathit{id}] : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathcal{D} \vdash \mathit{imports} : \langle \mathcal{T}_i, \mathcal{L}_i \rangle}{\mathcal{D} \vdash \textbf{import}\ \mathrm{A}\ \textbf{as}\ \mathrm{B}\ \mathit{imports} : \langle \mathcal{T}_i \trianglelefteq \mathcal{T}, \mathcal{L}_i \trianglelefteq \mathcal{L} \rangle} \quad \text{(T-Import-2)}$$

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \textbf{begin} : \langle \{\}, \{\} \rangle \quad \text{(T-Begin)}$$

$$\frac{\begin{array}{c}(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathrm{n} \notin \mathit{names}(\mathit{thyitems}) \\ \mathit{vars}(t) \subseteq \overline{\mathrm{X}} \qquad (\mathcal{T}, \mathcal{L}) \trianglelefteq (\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \vdash t\end{array}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{tac}\ \mathrm{n}(\overline{\mathrm{X}}) := t : \langle \mathcal{T}[\mathrm{n} \mapsto (\mathit{vis}, \overline{\mathrm{X}}, t)], \mathcal{L} \rangle} \quad \text{(T-Tac)}$$

$$\frac{\begin{array}{c}(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathrm{n} \notin \mathit{names}(\mathit{thyitems}) \\ \mathit{vars}(t) \subseteq \overline{\mathrm{X}} \qquad (\mathcal{T}, \mathcal{L}) \trianglelefteq (\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \vdash t\end{array}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{hitac}\ \mathrm{n}(\overline{\mathrm{X}}) := t : \langle \mathcal{T}[\mathrm{n} \mapsto (\mathit{vis}, \overline{\mathrm{X}}, [n]\ t)], \mathcal{L} \rangle} \quad \text{(T-Hitac)}$$

$$\frac{\begin{array}{c}(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathrm{n} \notin \mathit{names}(\mathit{thyitems}) \\ (\mathcal{T}, \mathcal{L}) \trianglelefteq (\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \vdash \mathit{prf}\end{array}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{lemma}\ \mathrm{n}\textbf{:}\ \gamma\ \mathit{prf} : \langle \mathcal{T}, \mathcal{L}[\mathrm{n} \mapsto (\mathit{vis}, \gamma, \mathit{id})] \rangle} \quad \text{(T-Lemma)}$$

Figure 5.10: Checking well-formedness of proof documents

The theory item rules check the same properties as basic theories except that well-formedness of tactics and lemmas is now defined with respect to the restricted environment, ensuring that private tactics and lemmas cannot be referred to outside the proof documents in which they are defined. Theory evaluation is then defined by the rules in Figure 5.11. A theory is evaluated at the top-level by the rule THY-

$$\frac{\mathcal{D} \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} : \langle \mathcal{T}', \mathcal{L}' \rangle \quad \mathcal{D} \vdash \mathit{imports} \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\mathcal{D} \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} \Downarrow \langle \mathcal{T} \lhd \mathcal{T}_i, \mathcal{L} \lhd \mathcal{L}_i \rangle} \text{(THY-EVAL)}$$

$$\frac{}{\mathcal{D} \vdash [] \Downarrow \langle \{\}, \{\} \rangle} \text{(IMPORT-EMPTY-EVAL)}$$

$$\frac{\mathcal{D} \vdash \mathcal{D}(A)[\mathit{id} := A.\mathit{id}] \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathcal{D} \vdash \mathit{imports} \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle}{\mathcal{D} \vdash \textbf{import}\ A\ \mathit{imports} \Downarrow \langle \mathcal{T}_i \lhd \mathcal{T}, \mathcal{L}_i \lhd \mathcal{L} \rangle} \text{(IMPORT-1-EVAL)}$$

$$\frac{\mathcal{D} \vdash \mathcal{D}(A)[\mathit{id} := B.\mathit{id}] \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathcal{D} \vdash \mathit{imports} \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle}{\mathcal{D} \vdash \textbf{import}\ A\ \textbf{as}\ B\ \mathit{imports} \Downarrow \langle \mathcal{T}_i \lhd \mathcal{T}, \mathcal{L}_i \lhd \mathcal{L} \rangle} \text{(IMPORT-2-EVAL)}$$

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \textbf{begin} \Downarrow (\{\}, \{\}) \text{(BEGIN-EVAL)}$$

$$\frac{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{tac}\ n(\overline{X}) := t \Downarrow \langle \mathcal{T}[n \mapsto (\mathit{vis}, \overline{X}, t)], \mathcal{L} \rangle} \text{(TAC-EVAL)}$$

$$\frac{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{hitac}\ n(\overline{X}) := t \Downarrow \langle \mathcal{T}[n \mapsto (\mathit{vis}, \overline{X}, [n]\ t)], \mathcal{L} \rangle} \text{(HITAC-EVAL)}$$

$$\frac{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad \langle \gamma, \mathit{prf} \rangle \Downarrow_{(\mathcal{T}_i \lhd \mathcal{T}, \mathcal{L}_i \lhd \mathcal{L})} \langle s \rangle}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \textbf{lemma}\ n\textbf{:}\ \gamma\ \mathit{prf} \Downarrow \langle \mathcal{T}, \mathcal{L}[n \mapsto (\mathit{vis}, \gamma, [n]\ s)] \rangle} \text{(LEMMA-EVAL)}$$

Figure 5.11: Evaluating proof documents

EVAL, which ensures the theory is well-formed then evaluates all the imports before evaluating the theory items. Individual theory items are evaluated similarly to the basic theory evaluation: tactics are simply added to the environment and lemmas are evaluated using the Hiscript evaluation relation. This time, however, lemmas are evaluated in the wider environment of the imported environment joined with the current theory environment. It is important to note that it does not use the restricted environment as was true for well-formedness checking. This is because evaluation of tactics will include unfolding public definitions that may be constructed with private definitions which therefore need to be in the environment. It is safe to do this because, already, well-formedness checking ensured that only public tactics and lemmas are referenced.

5.3.2.1 *Correctness of evaluation semantics*

As with basic theories, evaluation constructs well-formed proof environments:

Theorem 17 (Correctness of proof document evaluation)**.** *If*

$$\mathcal{D} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

then:

- *for every name, if* $\mathcal{L}(\mathit{name}) = (\gamma, s)$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*;*
- $\mathcal{L} \vdash \mathcal{T}$.

Proof. We take an identical approach as for basic theories (Theorem 15). Thus, we need to show that the environment constructed by the well-formedness judgement is also well-formed:

Lemma 18 (Correctness of proof document well-formedness)**.** *If*

$$\mathcal{D} \vdash \mathit{theory} : \langle \mathcal{T}, \mathcal{L} \rangle$$

then:

- *for every name, if* $\mathcal{L}(\mathit{name}) = (\gamma, s)$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g*;*
- $\mathcal{L} \vdash \mathcal{T}$.

Proof of lemma. Once again, every element of $\mathcal{L}$ has an identity proof therefore if $\mathcal{L}(\mathit{name}) = (\gamma, s)$ then $s \equiv \mathit{id} \vdash \gamma \longrightarrow \gamma$ by the hiproof validation rule V-Id.

To prove that $\mathcal{L} \vdash \mathcal{T}$, we note that the root rule in the derivation of well-formedness is T-Thy. Now, this tells us that

$$\mathcal{T} \equiv \mathcal{T}_i \Lsh \mathcal{T}'$$

where $\mathcal{T}'$ is the tactic environment constructed by the theory items. Similarly for the lemma environment. Now, if $\mathcal{T}'$ and $\mathcal{T}_i$ are well-formed, then their union is well-formed. Thus, we need to show that $\mathcal{L}' \vdash \mathcal{T}'$ and $\mathcal{L}_i \vdash \mathcal{T}_i$ with a simple induction. The nontrivial cases are (for T-Import-1) where we show:

$$\mathcal{D} \vdash \mathcal{D}(\mathsf{A})[\mathit{id} := \mathsf{A}.\mathit{id}] : \langle \mathcal{T}, \mathcal{L} \rangle$$

and similarly (for T-Import-2) where we show:

$$\mathcal{D} \vdash \mathcal{D}(\mathsf{A})[\mathit{id} := \mathsf{B}.\mathit{id}] : \langle \mathcal{T}, \mathcal{L} \rangle$$

where B is a fresh theory name in the current document. The well-formedness of these environments is exactly the property needed. Thus, we assume for a base case of the induction (the rule T-Import-Empty) that there is a theory without any imports and for that we only need to show $\mathcal{L}' \vdash \mathcal{T}'$.

To prove this, we need to induct on the theory item rules: that is, rules of the form

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle$$

where we assume $(\mathcal{T}_i, \mathcal{L}_i)$ is a well-formed environment. From here, the proof is similar to Theorem 15, except that we need to appeal to well-formedness of the environment union for the rule T-Lemma. □

This lemma can then be used directly to prove the theorem about evaluation. □

5.3.3 *Properties of theory evaluation*

This section extends the notions of minimal environment and environment extension to cover proof documents. Section A.2.1 of Appendix A details the properties proved about minimal environments and environment extension. While they are not of great interest on their own, these properties are used heavily in the correctness proofs for the refactoring specifications that are provided in Chapter 10 of Part 2.

5.3.3.1 *Minimal environments*

Just as it is possible to consider the minimal environment for an individual Hiscript proof or an individual hitac term, it is possible to consider a minimal environment for a whole theory. That is, the smallest imported environment to evaluate the whole theory. In analogy with excluding local definitions in proofs, both local definitions and *theory item definitions* are excluded in defining a minimal environment for a theory. This definition is expressed in terms of the theory items that a theory consists of:

Definition 12 (Minimal environment for theory items)**.** An imported environment $(\mathcal{T}_{min}, \mathcal{L}_{min})$ is a minimal environment for the theory items *thyitems* if

- $(\mathcal{T}_{min}, \mathcal{L}_{min}) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle$.
- For any well-formed environment $(\mathcal{T}_i, \mathcal{L}_i) \subset (\mathcal{T}_{min}, \mathcal{L}_{min})$, the theory items are not well-formed.

5.3.3.2 *Environment extension*

It is also possible to add more to an imported environment for a list of theory items:

Theorem 19 (Environment extension for theory items)**.** *Let thyitems be such that, for an environment* $(\mathcal{T}_i, \mathcal{L}_i)$*:*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

Then, for any well-formed environment $(\mathcal{T}'_i, \mathcal{L}'_i)$ *such that*

$$(\mathcal{T}_i, \mathcal{L}_i) \subseteq (\mathcal{T}'_i, \mathcal{L}'_i) \quad \textit{and} \quad \mathit{names}(\mathit{thyitems}) \cap (\mathit{names}(\mathcal{T}'_i) \cup \mathit{names}(\mathcal{L}'_i)) = \varnothing$$

then $(\mathcal{T}'_i, \mathcal{L}'_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$.

That is, one can extend an imported environment, so long as the extended environment does not contain any names that are introduced in the theory items (either as a lemma or tactic or as a *local* lemma or tactic inside a proof).

Theories are evaluated in the context of a *theory map*. It is natural to consider both:

- The smallest theory map that will successfully evaluate the theory.
- How can the theory map be extended in a way that ensures it still evaluates.

Both these properties are introduced in Section A.2.2 of Appendix A.

5.3.4 *Evaluating a proof document*

In order to get assurance that all the theories in a document are well-formed and evaluate successfully, I now introduce a method to evaluate and check well-formedness and evaluation of all theories, by means of the relations:

$$\vdash \mathcal{D} : \mathcal{E}$$

and

$$\vdash \mathcal{D} \Downarrow \mathcal{E}$$

where $\mathcal{D}$ is the theory map from above. I assume that an order has been attached to this map, and view it as a list. This order must respect the dependency structure of the imports for evaluation to succeed. The constructed map $\mathcal{E}$ maps each theory name to the environment that it constructs:

$$\mathcal{E} : \ \mathit{name} \rightarrow \mathit{proof\ environment}$$

The well-formedness rules that construct the *proof environment map* are shown below and the evaluation rules are identical, except they use the proof document theory evaluation relation. The conditions on the rules ensure that the theory map $\mathcal{D}$ doesn't contain any circular references or duplicated names. As with theory item well-formedness, the rules are given in a style for extending the theory map. I write $\varnothing$ to represent an empty map.

$$\frac{}{\vdash [] : \varnothing} \qquad \text{(PD-Empty)}$$

$$\frac{\mathfrak{n} \notin \mathit{thynames}(\mathit{thys}) \qquad \mathit{imports}(\mathit{thy}) \subseteq \mathit{thys} \qquad \vdash \mathit{thys} : \mathcal{E} \qquad \mathit{thys} \vdash \mathit{thy} : \langle \mathcal{T}_{\mathfrak{n}}, \mathcal{L}_{\mathfrak{n}} \rangle}{\vdash \mathit{thys}\ (\mathfrak{n}, \mathit{thy}) : \mathcal{E}[\mathfrak{n} \mapsto (\mathcal{T}_{\mathfrak{n}}, \mathcal{L}_{\mathfrak{n}})]} \qquad \text{(PD-Theory)}$$

The rule PD-Theory states that if the theory map *thys* is well-formed, then extending the theory map to include a theory *thy*, whose the name is $\mathfrak{n}$, will succeed if:

1. The name $\mathfrak{n}$ has not already been used: $\mathfrak{n} \notin \mathit{thynames}(\mathit{thys})$. The function *thynames* returns the set of all names of theories and also theories imported using the **import as** statement.
2. All the theories directly imported by *thy* already exist in the theory map.
3. Finally, the theory must be well-formed in the current theory map.

5.3.5 *Example document*

In this section, the basic theory shown in Figure 5.8 is split into a proof document consisting of multiple theories. The theories are shown in Figures 5.12 to 5.14.

From this example one can see the structuring power of this simple import system. The theory *tacticals*, for example, will be widely applicable and used many times over.

```
theory tacticals
begin

public tac ALL(X) := X ⊗ (ALL(X) | ⟨⟩ )
public tac ID := ALL(id)
public tac REPEAT(X) := X ;
    (ALL(REPEAT(X)) | ID)

...
end
```

```
theory setbasics
  import tacticals
begin

private tac intro := =—intro | ⊆—intro |
    ∩—intro
public hitac intros :=
     tacticals .REPEAT(intro)

public lemma emptysubany: ∅ ⊆ A
proof( intro )
  show x ∈ ∅ ⊢ x ∈ A
    by empty
qed

...
end
```

```
theory sets
  import setunion
  import setintersection
   ...
begin
...
end
```

Figure 5.12: Basic tactics and lemmas

```
theory setunion
  import setbasic
begin

public lemma unsubleft: A ⊆ A ∪ B
  by ⊆—intro ; ∪—intro1 ; ax

public lemma unsubright: B ⊆ A ∪ B
  by ⊆—intro ; ∪—intro2 ; ax

public lemma unempty: A ∪ ∅ = A
proof( setbasics . intros )
  show A ⊆ A ∪ ∅
    by lem unsubleft
  show A ∪ ∅ ⊆ A
  proof (∪—smallest)
    show A ⊆ A
      by ax
    show ∅ ⊆ A
      by setbasics .emptysubany
  qed
qed

public lemma AunA: A ∪ A = A
proof(=—intro)
  show AunAsubA: (A ∪ A) ⊆ A
  proof( setbasics . intros )
    show x ∈ (A ∪ A) ⊢ x ∈ A
      by ∪—elim ; tacticals .ALL(ax)
  qed
  show AsubAunA: A ⊆ (A ∪ A)
    by lem unsubleft
qed


...
end
```

Figure 5.13: Lemmas about set union

```
theory setintersection
  import setbasic
begin

public lemma intersubleft: A ∩ B ⊆ A
proof( setbasics . intros )
show x ∈ A ∩ B ⊢ x ∈ A
  by ∩−elim ; ax
qed

public lemma intersubright: A ∩ B ⊆ B
proof( tacticals . intros )
show x ∈ A ∩ B ⊢ x ∈ B
  by ∩−elim ; ax
qed

public lemma interempty: A ∩ ∅ = ∅
proof( setbasics . intros )
  show A ∩ ∅ ⊆ ∅
    by lem  intersubright
  show ∅ ⊆ A ∩ ∅
    by lem  setbasics .emptysubany
qed

public lemma AintA: A ∩ A = A
proof(=−intro)
  show A ∩ A ⊆ A
    by lem  intersubleft
  show A ⊆ A ∩ A
  proof (⊆−intro)
    show x ∈ A ⊢ x ∈ A ∩ A
      by ∩−intro ;  tacticals .ALL(ax)
  qed
qed

...
end
```

Figure 5.14: Lemmas about set intersection

The theory *setbasics* has the only private item of the document. It is used, primarily, to shorten the definition of *intros*, but there is also an instance of it being used in the proof of *emptysubany*. The reason this tactic definition is kept private is that explicit naming of the introduction rule used leads to more readable proofs. It could equally well be another public definition.

The tactic *intros* is used heavily in the theories that import *setbasics*: *setunion* and *setintersection*. For this tactic, the name qualification is useful as it can give similar tactics an identical name in their local theory, but they are named apart in descendants. Thus, one could have a theory:

```
theory fol
  import tacticals
begin

private tac intro = ∧—intro | ⇒—intro | ∀—intro ...
public hitac intros = tacticals .REPEAT(intro)

...
end
```

that defines a first order logic version. Any time both theories are in the proof environment, there are no name clashes as I refer to *fol.intros* and *setbasics.intros*. The theory *sets* in Figure 5.12 acts to combine together all the results about sets. This is the only theory in which evaluation will cause evaluation of all the theories in the document. Evaluation of *tacticals* will only evaluate that theory; evaluating *setbasics* will only evaluate *tacticals* as the sole import, before evaluating the contained theory items.

5.4 DISCUSSION

The introduction of theories and proof documents in this chapter completes the picture gradually painted in the past three chapters: a generic proof system that constructs hiproofs, where each part of the language has a precise semantics.

In this chapter, I introduced a notion of *theory* and gave it a semantics where evaluating a theory constructs a proof environment, which can be used for developing a larger theory. Hiscript theories — and environments — consist of tactics definitions and lemmas. Systems like Isabelle and Coq further allow for the introduction of conservative definitions as well as syntactic sugar and axioms. Hiscript could also be extended with these constructs by generalising a proof environment to be an n-tuple. Theories can be structured into a proof document and and imported by other theories, where an interface mechanism based on a *public* and *private* annotation controls the items that are imported. Hiscript uses a simple name qualification mechanism for ensuring that there are no name clashes during theory imports. This mechanism has a limitation that there can be no overriding of names or *opening* of a theory, but makes dependencies between theories explicit. This design decision is motivated by Hiscript's role as a simple language for investigating semantics for proof documents.

Hiscript also separates simple well-formedness checking from (possibly computationally intensive) proof evaluation in analogy with the two-step process of typing

checking and evaluation of a program. The well-formedness checks pick up static errors such as name clashes and tactics being applied to the wrong number of parameters.

In the next part of this thesis, I utilise this semantics heavily to give a formal assurance of correctness of structured transformations called *refactorings*.

6

AN ESSENCE OF SSREFLECT

6.1 INTRODUCTION

George Gonthier's SSReflect is a powerful language for proving theorems in the Coq system and was initially developed to facilitate his proof style of *small scale reflection* during the pioneering formalisation of the Four Colour Theorem (FCT) (Gonthier et al., 2008; Gonthier, 2008). Gonthier advocates a clear style of proof, which motivates many of the constructs in the language. A mixture of the declarative and procedural styles, SSReflect is designed to help write proofs that are readable upon replay; that is, one can write scripts where the chain of reasoning is clear when viewing the intermediate proof states. Using hiproofs, I give a semantics to eSSence, a subset of the SSReflect language that mostly focuses on proof style, with the dual aims of exploring the expressivity of the hiproof framework and clarifying the SSReflect style, which is by no means Coq-specific.

CHAPTER MAP This chapter presents the eSSence language. In the next section, I introduce the language with a simple example. I then describe the key elements of SSReflect *style* in Section 6.3 before formally introducing the syntax in Section 6.4. The formal semantics for the language is introduced in Sections 6.5 and 6.6. Finally, I conclude with a short summary in Section 6.8.

CONTRIBUTIONS This chapter's main contribution is a formal semantics for the eSSence language. The separate contributions are as follows:

1. The eSSence language is presented in a formal way.
2. Two formal semantics are given: a direct translation to Hitac and a big step evaluation relation. Furthermore, these two relations are proved to coincide.
3. The hierarchy in hiproofs is demonstrated to have a natural mapping to the annotation language of SSReflect and provides structure to the underlying proof.

This work was previously published in Whiteside et al. (2012).

6.2 INTRODUCING THE LANGUAGE

Since SSReflect differs from the procedural and declarative language seen before, I give some examples before delving into the formalities. Consider the simple eSSence (and SSReflect) proof of $(A \to B \to C) \to (A \to B) \to A \to C$, shown in Listing 6.1

```
move ⇒ h1 h2 h3.
move: h1.
apply.
  by [].
apply: h2.
by [].
```

Listing 6.1: A proof of $(A \to B \to C) \to (A \to B) \to A \to C$ in eSSence

and a more concise variant in Listing **??**. The underlying logic is that given in Section 3.2.4.2.

An eSSence proof is a *paragraph* that, in the case of Listing 6.1, consists of three *sentences* and then two nested paragraphs (the first is a single sentence long, the second is two sentences long). The indentation marks the start of the first branch, and the second branch is at the same level as the initial goal. If more than two goals are introduced, *annotations* are used to explicitly show where the proof has branched. Each sentence evaluates on the first goal in a goal stack and any subgoals resulting from the execution of the sentence are pushed back on the top of the stack. Each sentence is explained briefly:

1. The **move** tactic plays an explanatory role (acting as the identity tactic), indicating that assumptions and variables are being moved to/from the context. The work is done by ':' and '⇒', which are actually tacticals to *pop* and *push* from/to the proof context. This first sentence extends the context to:

   $h1 : (A \to B \to C)$
   $h2 : (A \to B)$
   $h3 : A$
   ==========
   C

2. The sentence **move**: h1. will then *push* h1 back into the goal, transforming it to $(A \to B \to C) \to C$. The full context then looks like:

   $h2 : (A \to B)$
   $h3 : A$
   ==========
   $(A \to B \to C) \to C$

3. The **apply** tactic, the standard backwards proof tactic in Coq, attempts to match the conclusion of the goal with the conclusion of the first assumption — moti-

vating the previous step — then breaks down the assumption as new subgoals which, in this case, are A and B. After this step, there are two goals:

```
h2 : (A → B)
h3 : A
================
A                    (1/2)
B                    (2/2)
```

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 default automation. In this case, the assumption h3 is used. After solving A, the context looks like:

```
h2 : (A → B)
h3 : A
================
B
```

5. The second branch requires an application and a further use of the by tactical. In this case, 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, resulting in a context:

```
h3 : A
================
A
```

which is solved by the default automation of the by tactical as before.

6.3 SSREFLECT STYLE

SSReflect comes with a clear idea of proof style. Indeed, a large part of the language — the part that eSSence focuses on — facilitates a style of proof that is designed to create proof scripts that are *robust*, *maintainable*, and *replayable*. This section describes the main points of SSReflect style, as made clear in conversations with Gonthier (Gonthier, 2012) and in the SSReflect reference manual (Gonthier et al., 2008).

- Proofs in SSReflect should be developed to ensure that each line (or *sentence*) has a clear meaning mathematically (related formula manipulations, an inductive step, etc). This helps make the scripts *readable upon replay*.
- The overall branching structure of a proof should be made clear by annotations and indentations. The SSReflect annotation guidelines state that:

  - If a sentence leaves one subgoal, then no indentation or annotation is required. If a tactic sentence introduces two subgoals — the apply tactic in the above example — then the proof of the first goal is indented. The second is at the same level of indentation as the parent goal.
  - If a sentence introduces three or more subgoals then bullets and indentation are required to mark the start of the proof of each subgoal. 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.

- In order to make the most important subgoal last, SSReflect provides tacticals to rotate the order of subgoals and solve goals 'inline' to help achieve this.
- Anywhere that a goal is solved, the *closing* tactical by should be used. This increases the robustness and maintainability of the script. If, upon proof replay, the tactic no longer solves the goal, it will be immediately flagged up by failure of this tactical.
- SSReflect has specific tacticals for performing bookkeeping operations, which force names to be explicitly provided for any variables and assumptions introduced into the context. Furthermore, the importance of choosing good names is emphasised as crucial to a maintainable proof.

In the example in the previous section, two main features of the language can be seen: clustering all bookkeeping operations into two tacticals (: and =>) and providing structure and robustness to scripts using indentation, annotation, and by. However, this isn't a very well-presented script. It is, perhaps, too verbose and the assumption names do not convey any information. As a first step, the hypotheses can be renamed to convey meaning. For example renaming h1 to hAiBiC. The proof can then be improved by compressing move: hAiBiC. and apply. into a single line, using the THEN tactical (;), resulting in move: hAiBiC ; apply.. In fact, the semantics for the language equates this to apply: hAiBiC since move behaves as an identity tactic.

Finally, the first tactical can be used to compress the two lines:

```
apply: hAiBiC.
  – by [].
```

into one:

```
apply: hAiBiC; first by [].
```

This tactical takes two tactics as a parameter, for example:

$tactic_1$; first $tactic_2$

and applies the first tactic to the current goal; then, the second tactic is applied to the first of the resulting subgoals *before* they are placed on the stack. Thus, the result of applying apply: hAiBiC; first by []. is the context:

```
hAiB : (A → B)
hA : A
================
B
```

The idea of this tactical is to solve simple goals or side-conditions before they become visible to the user in the proof context and helping keep the script linear. (There are more sophisticated variants of this tactical where the goal to be 'inlined' is not the first or to discharge multiple side-conditions in one fell swoop.) The resulting script after these changes is given in Listing **??**.

```
move => hAiBiC hAiB hA.
apply: hAiBiC; first by [].
by apply: hAiB.
```

6.4 SYNTAX OF THE LANGUAGE

The syntax for eSSence is given by the grammar in Figure 6.1. The idea, described above, is that eSSence scripts consist of *sentences* and *paragraphs*, given as separate syntactic classes in the grammar.

SENTENCES A *sentence* is a grammar element *sstac*. The parameters *num*, *term*, and *ident* stand for numerals, terms, and identifiers respectively. Each *sstac* is described below:

move will do nothing if an introduction step is possible; if not, it will attempt to reduce the goal to *head normal form*. The purpose of this tactic is mostly explanatory and is almost always used in combination with the *introduction* and *discharge* tacticals.

apply is the tactic for backwards proof in SSReflect. It is used in two forms in eSSence: on its own and in combination with the discharge tactical. In its basic form, it attempts to match the conclusion of the current goal with the conclusion of the first assumption of that goal. If the assumption matches exactly the current goal, then there are no goals generated; where it doesn't match, subgoals are generated. The subgoals generated are found by attempting to apply the Coq *refine* tactic with different amounts of *padding* (with proof variables) applied after the term until it fits.

rewrite allows for multiple, directed rewrite rules to be applied in a left to right order. A rewrite rule is simply a term that is matched against the current goal. I do not give a detailed specification of the matching process, but assume that it behaves appropriately. Gonthier, in the SSReflect reference manual describes

eSSence syntax

ssscript	::=	*sspara*	A proof script is a paragraph.
ssanno	::=	$+ \mid - \mid *$	Annotations can be these symbols.
sspara	::=	*sstac*.	A paragraph is a non-empty
		⋮	list of eSSence tactics
		sstac.	followed by zero or more
		(*ssanno sspara*)*	annotated paragraphs.
sstac	::=	*dtactic*	A tactic can be a discharge tactic;
	|	**rewrite** *rstep*$^+$	a series of rewrite steps;
	|	**have:** *term* (**by** *sstac*)$^?$	a forward step;
	|	*sstac* ; *chtac*	a THEN tactical;
	|	**by** *sstac*	a closing tactical;
	|	**exact** *term*	an exact proof term;
	|	*sstac* ; **first** *num*$^?$ *ochtac*	solve a subset of goals;
	|	*sstac* ; **last** *num*$^?$ *ochtac*	solve a subset from the back;
	|	*sstac* ; **first** *num*$^?$ **last**	rotate the goals left;
	|	*sstac* ; **last** *num*$^?$ **first**	rotate the goals right;
	|	*dtactic* : *ditem* ... *ditem*	be a *discharge* tactical;
	|	*sstac* ⇒ *iitemstart*$^?$ *iitem* ... *iitem*	or be an *introduction* tactical.
dtactic	::=	**move** | **apply**	Discharge tactics are *move* or *apply*.
chtac	::=	*sstac*	A choice expression is a tactic
	|	[*sstac* | ... |*sstac*]	or a list of possible tactics.
ochtac	::=	*sstac*	In an optional choice expression
	|	[*sstac*$^?$ | ... |*sstac*$^?$]	elements of the list can be blank.
iitemstart	::=	*iitem* | [$iitem_1^*$ | ... |$iitem_n^*$]	Introduction item lists.
iitem	::=	*sitem* | *ipattern*	An introduction item is a pattern.
ipattern	::=	**ipatt** *ident*	Patterns are simply identifiers.
rstep	::=	(-$^?$*term*) | *sitem*	Rewrite steps either use a term or...
sitem	::=	`/=` | `//` | `//=`	apply simplification procedures.
ditem	::=	*term*	Simply a term to discharge.

Figure 6.1: Syntax for eSSence

several matching algorithms used in SSReflect (Gonthier et al., 2008). As well as rewrite rules, so-called *simplification items* (*sitem*) can be applied in the middle of rewriting. There are three simplification items:

1. / = applies a simplification tactic, which operates by partially evaluating the goal;
2. // solves trivial goals using a tactic called *DONE*: see the **by** tactical description for more information about this tactic;
3. // = combines the above simplification tactics.

$sstac_1$; $sstac_2$ is the first form of the eSSence THEN tacticals and corresponds exactly to the LCF THEN tactical.

$sstac$; $[sstac_1 | \ldots | sstac_n]$ is the *branching THEN* tactical, also known as THENL in HOL terminology. It applies the first tactic, $sstac$, generating a list of n subgoals each of which has an element in the tactic list. If the number of tactics in the list isn't equal to the number of generated goals, then this tactic fails.

by $sstac$ is the closing tactical in SSReflect, also called the *terminating* tactical. The supplied tactic $sstac$ is applied to the current goal, followed by some simple automation in the form of a tactic called *DONE*. If this doesn't solve the goal completely, then the tactical fails. The *DONE* tactic is left undefined in eSSence, but can provide some basic automation for solving simple goals.

exact $term$ will solve the goal if the supplied term has the same type as the current goal; otherwise it will fail.

have : $term$ introduces a new fact as an assumption to the current goal, which can be used for forward proof. The supplied $term$ is introduced as an assumption to the current goal — explaining why there is no name supplied: it can be introduced later if required — and another subgoal is generated requiring a proof of the new fact. If the optional **by** $sstac$ is also supplied then it must prove the newly introduced fact.

$sstac_1$; **first** k $sstac_2$ is the basic form of the **first** tactical and will apply $sstac_2$ to the kth subgoal generated by $sstac_1$. It will fail if the numbers do not match. The number is optional and, if omitted, it will default to 1.

$sstac$; **first** k $[sstac_1 | \ldots | sstac_m]$ is an extended form of the **first** tactical and will apply $sstac_1$ to the kth goal generated by $sstac$, $sstac_2$ to the k + 1th, etc. The tactics in the branching construct are optional:

$sstac$; **first** $[sstac_1 | | | sstac_2]$

will apply $sstac_1$ to the first subgoal and $sstac_2$ to the fourth subgoal and leave the second and third (and fifth etc.) unchanged.

$sstac_1$; **last** k $sstac_2$ is the basic form of the **last** tactical and will apply $sstac_2$ to the kth subgoal *from the end* of the goals generated by $sstac_1$. It will fail if the numbers do not match. As before, the number parameter is optional. This time, though, the default will apply the tactic to the last subgoal.

sstac ; **last** k [$sstac_1$|...|$sstac_m$] is the extended form of the **last** tactical. It behaves similarly to **first**; however, this time the tactic $sstac_m$ is applied to the kth goal, $sstac_{m-1}$ is applied to the $k-1$th goal and so on. The tactic:

sstac ; **last** [$sstac_1$ | | | $sstac_2$]

will apply $sstac_2$ to the final subgoal and $sstac_1$ to the fourth from final subgoal leaving the rest unchanged.

sstac ; **first** k **last** is a tactical to perform *goal rotation*. The tactic can be applied without a parameter:

sstac ; **first last**

and will invert the order of goals. An application with a parameter supplied:

sstac ; **first** k **last**

will rotate the goals so that the first goal becomes the kth goal. That is, it will perform a one-step rotation to the right $k-1$ times.

sstac ; **last** k **first** rotates goals in the other direction. Without a parameter, k, **last first** behaves identically to **first last** and inverts the order of goals. With the k present, it will perform $k-1$ rotations to the left. For example, applying

sstac ; **last** 3 **first**

to the list of goals $[\gamma_1, \gamma_2, \gamma_3, \gamma_4]$ would result in $[\gamma_3, \gamma_4, \gamma_1, \gamma_2]$.

move : $term_1 \ldots term_n$ is the first form of the *discharge* tactical in eSSence. The discharge tactical in eSSence takes a list of terms and discharges them from right to left ($term_n$ first) one at a time before applying the supplied tactic. Discharging removes the term from the context and abstracts over it, creating a new assumption of the goal. For example, applying the tactic

move: hQ hP

to a goal `hP` : P, `hQ` : $Q \vdash R$ would result in a goal $\vdash Q \to P \to R$.

apply : $term_1 \ldots term_n$ is the second form of the discharge tactical. It will attempt to use the **apply** tactic with the term $term_1(term_2 \ldots term_n)$. If this term does not match the goal, padding is added between $term_1$ and the rest of the applied terms. To understand this, consider a context:

```
f : ∀x, x < 5 → prime x → x/4 = 0
lty5 : y < 5
y : nat
prx : prime x
================
y/4 = 0
```

Now, the following tactic:

apply: f lty5 prx

will *not* revert `f` and `lty5` to give the context:

$$y : nat$$
$$==========$$
$$(\forall\ x,\ x < 5 \to prime\ x \to x/4 = 0) \to y < 5 \to prime\ x \to y/4 = 0$$

but rather, attempt to apply `f lty5 prx` to the goal. This is not a direct match, however, so the **apply** tactic attempts to pad the term as `f ?x lty5 prx`, which does succeed.

$sstac \Rightarrow ident_1 \ldots ident_n$ is the simplest instance of the *introduction* tactical. The tactic $sstac$ is executed, then the basic Coq *intro* tactic is applied for each identifier in left to right order. Simplification items can also be intermingled with the identifiers.

$sstac \Rightarrow [ident_1^1 \ldots ident_1^n | \ldots | ident_m^1 \ldots ident_m^{n'}]$ is the branching version of the introduction tactical. A branching pattern can only be the first introduction item in any instance of the introduction tactical. The number of branches must correspond to the number of subgoals generated. For example, consider the goal

$$(P \to Q) \wedge (A \to B \to C)$$

which can be broken down by the tactic

apply conjE $\Rightarrow$ [hP | hA hB]

For a full presentation and a tutorial guide to the original SSReflect language, see Gonthier et al. (2008); Gonthier and Roux (2009).

PARAGRAPHS A proof script in SSReflect is a list of sentences, separated by full stops; however, one can optionally *annotate* a script, which, along with indentation, helps make clear the subgoal flow within a script, as described in Section 6.3. The idea of these annotations is to use bullets ($*$, $+$, and $-$) to highlight where a proof branches. This is built directly into eSSence 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, each must be annotated identically. In eSSence, the annotations are simplified 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 6.2 contains examples of scripts that follow these structuring guidelines (on the right is an indication of the arity of each tactic).

6.5 SENTENCE SEMANTICS

The semantics for eSSence sentences is based on a big step relation on tactics:

$$\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$$

		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.	$[\gamma_4] \to []$

Figure 6.2: A linear script, one level of branching, and multiple branching levels.

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'. The evaluation rules are presented in Figures 6.3 and 6.4. For brevity, I will explain the rules in Figure 6.3 and leave the rest for the reader. Note that the Hitac syntax is *lifted* to the level of eSSence tactics for the evaluation. For example, this allows evaluation of ; — the THEN tactical — to be expressed directly as $sstac_1$; $ALL(sstac_2)$. A small set of predefined tactics and tacticals — all with uppercase names — is also assumed, many of which are defined in Section 3.3.5 on page 41. Recall also that the atomic tactics for eSSence are defined in Section 3.2.4.2 on page 31. Those that are not are now described briefly:

HNF reduces a goal to *head normal form*. That is, it reduces the head of the goal until it becomes a product or an irreducible term.

DONE performs some unspecified simplification that generally 'solves trivial subgoals'. SSReflect uses the Coq *trivial* tactic (The Coq development team, 2004).

SIMP also performs unspecified simplification, but is a more powerful tactic than *DONE*. In SSReflect, it involves computation to simplify the goal.

REWRITE is a tactic that performs a single rewrite step.

The notation $t^n_\otimes$ means a tensor product of length n and similarly for sequencing. Thus, $id^n_\otimes$ means the tensor product $id \otimes \ldots \otimes id$.

SS-MOVE. The **move** tactic behaves as an identity if an introduction step is possible or transforms the goal to head normal form otherwise. This is modelled by providing a Hitac assertion for checking that the goal is of a product form and applying the *HNF* tactic, which reduces the goal to head normal form. Note also that the application of **move** is 'boxed up' to abstract away from any normalisation done by this tactic. This is the first of many occasions in the semantics where hierarchy is used to abstract away details and add structure to the underlying proof.

SS-BY. This rule evaluates the closing tactical. It first applies the supplied tactic $sstac$ then, to all the subgoals generated, applies the *DONE* tactic. The empty tactic $\langle\rangle$ is used to fail this tactic if there are subgoals remaining.

SS-FIRST-1. In this primitive form of selection tactical, the tactic $sstac_2$ is only applied to the first subgoal generated from evaluating $sstac_1$ and hierarchy is added to encapsulate the proof. The hiproof that is constructed uses identities to ignore the $n - 1$ goals not operated on.

$$\frac{\langle \gamma, assert\ (\Gamma \vdash \Pi x : A.B) \mid HNF\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{move}\rangle \Downarrow_{\mathcal{E}} \langle [move]\ s, g\rangle} \quad \text{(SS-Move)}$$

$$\frac{\langle \gamma, sstac_\rangle \Downarrow_{\mathcal{E}} \langle s_1, g\rangle \qquad \langle g, ALL(DONE); \langle\rangle\rangle \Downarrow_{\mathcal{E}} \langle s_2, []\rangle}{\langle \gamma, \textsf{by}\ sstac\rangle \Downarrow_{\mathcal{E}} \langle [by]\ s_1\ ;\ s_2, []\rangle} \quad \text{(SS-By)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, [\gamma_1, \ldots, \gamma_n]\rangle \qquad \langle \gamma_1, ([first]\ sstac_2)\rangle \Downarrow_{\mathcal{E}} \langle s_2, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{first}\ sstac_2\rangle \Downarrow_{\mathcal{E}} \langle s_1\ ;\ (s_2 \otimes id_{\otimes}^{n-1}), g@[\gamma_2, \ldots, \gamma_n]\rangle} \quad \text{(SS-First-1)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_\gamma\rangle \qquad \langle g_\gamma, REFLECT\rangle \Downarrow_{\mathcal{E}} \langle s_2, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{first last}\rangle \Downarrow_{\mathcal{E}} \langle [FL]\ s_1\ ;\ s_2, g\rangle} \quad \text{(SS-FirstLast-1)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_\gamma\rangle \qquad \langle g_\gamma, ROTATE_R^{k-1}\rangle \Downarrow_{\mathcal{E}} \langle s_2, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{first k last}\rangle \Downarrow_{\mathcal{E}} \langle [FkL]\ s_1\ ;\ s_2, g\rangle} \quad \text{(SS-FirstLast-2)}$$

$$\frac{\begin{array}{c}\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_1]\rangle \\ \langle \gamma_1, iitem_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, [\gamma_2]\rangle \ldots \langle \gamma_n, iitem_n\rangle \Downarrow_{\mathcal{E}} \langle s_n, g_n\rangle\end{array}}{\langle \gamma, sstac \Rightarrow\ iitem_1 \ldots iitem_n\rangle \Downarrow_{\mathcal{E}} \langle [\Rightarrow]\ s\ ;\ s_1\ ;\ \ldots\ ;\ s_n, g\rangle} \quad \text{(SS-Intro-1)}$$

$$\frac{\langle \gamma, INTRO(ident)\rangle \Downarrow_{\mathcal{E}} \langle s, \gamma'\rangle}{\langle \gamma, \textsf{ipatt}\ ident\rangle \Downarrow_{\mathcal{E}} \langle s, \gamma'\rangle} \quad \text{(SS-IPatt)}$$

$$\frac{\langle \gamma, ASSERT(term)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{have:}\ term\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle} \quad \text{(SS-Have-1)}$$

$$\frac{\langle \gamma, sstac_1\ ;\ ALL(sstac_2)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, sstac_1 ; sstac_2\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle} \quad \text{(SS-Then)}$$

$$\texttt{//} \equiv ALL([DONE]\ \ DONE) \quad \text{(SS-Done)}$$

$$\texttt{/=} \equiv ALL([SIMP]\ \ SIMP) \quad \text{(SS-Simp)}$$

$$\texttt{//=} \equiv \texttt{/=}\ ;\ \texttt{//} \quad \text{(SS-DoneSimp)}$$

Figure 6.3: eSSence evaluation semantics (1)

$$\frac{\langle \gamma, ASSERT(term)\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_1, \gamma_2]\rangle \qquad \langle \gamma_1, \textsf{by } sstac\rangle \Downarrow_{\mathcal{E}} \langle s_{\gamma_1}, []\rangle}{\langle \gamma, \textsf{have: } term \textsf{ by } sstac\rangle \Downarrow_{\mathcal{E}} \langle [have]\ s\ ;\ (s_{\gamma_1} \otimes id), \gamma_2\rangle} \text{(SS-HAVE-2)}$$

$$\frac{\langle \gamma, sstac\ ;\ (sstac_1 \otimes \ldots \otimes sstac_n)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, sstac;\ [sstac_1 | \ldots | sstac_n]\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle} \text{(SS-THENLIST)}$$

$$\frac{\langle \gamma, INTRO(top)\ ;\ (REFINE(top) \mid REFINE(top\ \_) \mid \ldots)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{apply}\rangle \Downarrow_{\mathcal{E}} \langle [apply]\ s, g\rangle} \text{(SS-APPLY)}$$

$$\frac{\langle \gamma, EXACT(t)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{exact}(t)\rangle \Downarrow_{\mathcal{E}} \langle [exact]\ s, g\rangle} \text{(SS-EXACT)}$$

$$\frac{\langle \gamma, REWRITE(rev, tm)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{rstep } rev\ tm\rangle \Downarrow_{\mathcal{E}} \langle [rstep]\ s, g\rangle} \text{(S-RSTEP-SINGLE)}$$

$$\frac{\langle \gamma_1, rstep_1\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_2]\rangle \ldots \langle \gamma_n, rstep_n\rangle \Downarrow_{\mathcal{E}} \langle s_n, g_n\rangle}{\langle \gamma_1, \textsf{rewrite } rstep_1 \ldots rstep_n\rangle \Downarrow_{\mathcal{E}} \langle [rewrite]\ s_1\ ;\ \ldots\ ;\ s_n, g_n\rangle} \text{(SS-REWRITE)}$$

$$\frac{\begin{array}{c}\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad k + m \leqslant n \\ \langle \gamma_k, [first]\ sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_1\rangle \quad \ldots \quad \langle \gamma_{k+m}, [first]\ sstac_m\rangle \Downarrow_{\mathcal{E}} \langle s_m, g_m\rangle\end{array}}{\begin{array}{c}\langle \gamma, sstac\ ;\ \textsf{first } k\ [\ sstac_1 \mid \ldots \mid sstac_m\ ]\rangle \\ \Downarrow_{\mathcal{E}} \langle s\ ;\ id_{\otimes}^{k-1} \otimes s_1 \otimes \ldots \otimes s_m \otimes id_{\otimes}^{n+1-k-m}, \\ [\gamma_1, \ldots, \gamma_{k-1}]@g_1 \ldots g_m@[\gamma_{k+1+m}, \ldots, \gamma_n]\rangle\end{array}} \text{(SS-FIRST-2)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad \langle \gamma_n, [last]\ sstac_2\rangle \Downarrow_{\mathcal{E}} \langle s_{\gamma_n}, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{last } sstac_2\rangle \Downarrow_{\mathcal{E}} \langle s\ ;\ (id_{\otimes}^{n-1} \otimes s_{\gamma_n}), [\gamma_1, \ldots, \gamma_{n-1}]@g\rangle} \text{(SS-LAST-1)}$$

$$\frac{\begin{array}{c}\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad m \leqslant k \leqslant m \\ \langle \gamma_k, [last]\ sstac_m\rangle \Downarrow_{\mathcal{E}} \langle s_m, g_m\rangle \quad \ldots \quad \langle \gamma_{k-m}, [last]\ sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_1\rangle\end{array}}{\begin{array}{c}\langle \gamma, sstac\ ;\ \textsf{last } k\ [\ sstac_1 \mid \ldots \mid sstac_m\ ]\rangle \\ \Downarrow_{\mathcal{E}} \langle s\ ;\ id_{\otimes}^{k-m} \otimes s_1 \otimes \ldots \otimes s_m \otimes id_{\otimes}^{n-k}, \\ [\gamma_1, \ldots, \gamma_{k-m}]@g_1 \ldots g_m@[\gamma_{k+1}, \ldots, \gamma_n]\rangle\end{array}} \text{(SS-LAST-2)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_\gamma\rangle \qquad \langle g_\gamma, REFLECT\rangle \Downarrow_{\mathcal{E}} \langle s_2, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{last first}\rangle \Downarrow_{\mathcal{E}} \langle [LF]\ s_1\ ;\ s_2, g\rangle} \text{(SS-LASTFIRST-1)}$$

$$\frac{\langle \gamma, sstac_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, g_\gamma\rangle \qquad \langle g_\gamma, ROTATE_L^{k-1}\rangle \Downarrow_{\mathcal{E}} \langle s_2, g\rangle}{\langle \gamma, sstac_1\ ;\ \textsf{last } k \textsf{ first}\rangle \Downarrow_{\mathcal{E}} \langle [LkF]\ s_1\ ;\ s_2, g\rangle} \text{(SS-LASTFIRST-2)}$$

$$\frac{\langle \gamma, REVERT(term)\rangle \Downarrow_{\mathcal{E}} \langle s, \gamma'\rangle}{\langle \gamma, \textsf{ditem } term\rangle \Downarrow_{\mathcal{E}} \langle s, \gamma'\rangle} \text{(SS-DITEM)}$$

$$\frac{\begin{array}{c}\langle \gamma, \textsf{ditem } ditem_n\rangle \Downarrow_{\mathcal{E}} \langle s_n, \gamma_n\rangle \ldots \langle \gamma_2, \textsf{ditem } ditem_1\rangle \Downarrow_{\mathcal{E}} \langle s_1, \gamma_1\rangle \\ \langle \gamma_1, dtactic\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle\end{array}}{\langle \gamma, dtactic\textsf{: } ditem_1 \ldots ditem_n\rangle \Downarrow_{\mathcal{E}} \langle s_n\ ;\ \ldots\ ; s_1\ ;\ s, g\rangle} \text{(SS-DISCHARGE)}$$

$$\frac{\langle \gamma, (REFINE(t_1(t_2 \ldots t_n)) \mid REFINE(t_{1\_}\ (t_2 \ldots t_n)) \mid \ldots)\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle}{\langle \gamma, \textsf{apply : } t_1 \ldots t_n\rangle \Downarrow_{\mathcal{E}} \langle [apply]\ s, g\rangle} \text{(SS-APPLYDIS)}$$

Figure 6.4: eSSence evaluation semantics (2)

SS-FIRSTLAST-1. This tactical reflects the subgoals and is implemented by the hitac reflection tactic described in Section 3.3.5.

SS-FIRSTLAST-2. This rule is explained by an example application to a list of subgoals $[g_1, g_2, g_3, g_4, g_5]$. Assuming these goals are generated by $sstac_1$ in the tactic $sstac_1$; first 2 last, then performing the rotation would result in the remaining goals looking like $[g_3, g_4, g_5, g_1, g_2]$. The evaluation rule performs the appropriate number of rotations using a basic Hitac rotation tactic.

SS-INTRO-1 AND SS-IPATT. These rules make up the non-branching version of the introduction tactical, which looks like $sstac$=> $iitem_1 \ldots iitem_n$. An example instance of this tactical would be

```
move ⇒ hP hQ.
```

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 (as shown in the rule SS-IPATT). Recall that the atomic tactic *INTRO* will fail if the supplied identifier is not fresh in the context.

SS-HAVE-1. This rule evaluates a forward step and is directly given by the atomic tactic *ASSERT*, which is described in Section 3.2.4.2 on page 31.

SS-THEN. The eSSence *THEN* tactical is given simply the analogous hitac tactical.

SS-DONE TO SS-DONESIMP. The simplification items are implemented directly by Hitac tactics *DONE* and *SIMP* being applied to each goal in the current proof context.

Theorem 20 (Correctness of sentence evaluation)**.** *If*

$$\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$$

then

$$s \vdash \gamma \to g.$$

Rather than give a direct proof, I note that correctness of these evaluation rules follows from Lemma 21 and Lemma 22 from the next section.

6.5.1 *A direct translation to Hitac*

Each sentence can be statically translated to a hitac that can be evaluated directly:

$$\langle \gamma, [\![sstac]\!] \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g \rangle$$

where $[\![sstac]\!]$ is defined inductively in Figure 6.5.

Lemma 21 (Correctness of direct translation)**.** *If* $\langle \gamma, [\![sstac]\!] \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g \rangle$ *then* $s \vdash \gamma \to g$.

Proof. This is a direct result of correctness of hitac evaluation. □

$$
\begin{array}{ll}
[\![\textbf{move}]\!] & = [move]\ \ assert\ (\Gamma \vdash \Pi x : A.B) \mid HNF \\
\\
[\![\textbf{apply}]\!] & = [apply]\ \ INTRO(top)\ ; \\
 & \quad (REFINE(top) \mid REFINE(top\ \_) \mid \ldots) \\
[\![\textbf{apply}: t_1 \ldots t_n]\!] & = [apply]\ \ (REFINE(t_1(t_2 \ldots t_n)) \\
 & \quad \mid REFINE(t_{1\_}\ (t_2 \ldots t_n)) \mid \ldots) \\
[\![\textbf{exact}\ term]\!] & = [exact]\ \ EXACT(term) \\
\\
[\![\textbf{rewrite}\ rstep_1 \ldots rstep_n]\!] & = [rewrite]\ \ [\![rstep_1]\!]\ ;\ \ldots\ ;\ [\![rstep_n]\!] \\
[\![\textbf{rstep}\ rev\ term]\!] & = [rstep]\ \ REWRITE(rev, term) \\
\\
[\![\textbf{by}\ sstac]\!] & = [by]\ \ [\![sstac]\!]\ ;\ ALL(DONE)\ ;\ \langle\rangle \\
\\
[\![sstac_1\ ;\ \textbf{first}\ sstac_2]\!] & = [\![sstac_1]\!]\ ;\ ([first]\ \ [\![sstac_2]\!]) \otimes ID \\
\\
[\![sstac\ ;\ \textbf{first}\ k\ [sstac_1|\ldots|sstac_m]]\!] & = [\![sstac]\!]\ ;\ id_{\otimes}^{k-1}\ \otimes ([first]\ [\![sstac_1]\!]) \otimes \ldots \otimes ([first]\ [\![sstac_m]\!]) \otimes ID \\
\\
[\![sstac_1\ ;\ \textbf{last}\ sstac_2]\!] & = [\![sstac_1]\!]\ ;\ ID \otimes ([last]\ \ [\![sstac_2]\!]) \\
\\
[\![sstac\ ;\ \textbf{last}\ k\ [sstac_1|\ldots|sstac_m]]\!] & = [\![sstac]\!]\ ;\ id_{\otimes}^{k-m}\ \otimes ([last]\ [\![sstac_1]\!]) \otimes \ldots \otimes ([last]\ [\![sstac_m]\!]) \otimes ID \\
\\
[\![sstac\ ;\ \textbf{first last}]\!] & = [FL]\ \ [\![sstac]\!]\ ;\ REFLECT \\
\\
[\![sstac\ ;\ \textbf{first}\ k\ \textbf{last}]\!] & = [FkL]\ \ [\![sstac]\!]\ ;\ ROTATE_{R;}^{k-1} \\
\\
[\![sstac\ ;\ \textbf{last first}]\!] & = [LF]\ \ [\![sstac]\!]\ ;\ REFLECT \\
\\
[\![sstac\ ;\ \textbf{last}\ k\ \textbf{first}]\!] & = [LkF]\ \ [\![sstac]\!]\ ;\ ROTATE_{L;}^{k-1} \\
\\
[\![\textbf{have}: term]\!] & = ASSERT(term) \\
[\![\textbf{have}: term\ \textbf{by}\ sstac]\!] & = ASSERT(term)\ ;\ ([\![sstac]\!] \otimes id) \\
\\
[\![sstac_1\ ;\ sstac_2]\!] & = [\![sstac_1]\!]\ ;\ ALL(\ [\![sstac_2]\!]) \\
[\![sstac\ ;\ [sstac_1|\ldots|sstac_n]]\!] & = [\![sstac]\!]\ ;\ ([\![sstac_1]\!] \otimes \ldots \otimes [\![sstac_n]\!]) \\
\\
[\![sstac\texttt{=>}\ iitem_1 \ldots iitem_n]\!] & = [\texttt{=>}]\ \ [\![sstac]\!]\ ;\ [\![\textbf{ipatt}\ iitem_1]\!]\ ;\ \ldots\ ;\ [\![\texttt{ipatt}\ iitem_n]\!] \\
[\![\textbf{ipatt}\ ident]\!] & = INTRO(ident) \\
\\
[\![dtactic: ditem_1 \ldots ditem_n]\!] & = [:]\ \ [\![\textbf{ditem}\ term_n]\!]\ ;\ \ldots\ ;\ [\![\textbf{ditem}\ term_1]\!]\ ;\ [\![dtactic]\!] \\
[\![\textbf{ditem}\ term]\!] & = REVERT(term) \\
\\
[\![//]\!] & = ALL([DONE]DONE) \\
[\![/=]\!] & = ALL([SIMP]SIMP) \\
[\![//=]\!] & = [\![/=]\!]\ ;\ [\![//]\!]
\end{array}
$$

Figure 6.5: Direct translation of eSSence to hitac

Lemma 22 (Equivalence of evaluation relations). *For a sentence sstac applied to a goal γ if*

$$\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$$

then

$$\langle \gamma, [\![sstac]\!] \rangle \Downarrow^{t}_{\mathcal{E}} \langle s, g \rangle.$$

Proof. Proceed by induction on the height of the derivation. The base cases: SS-MOVE, SS-IPATT, SS-HAVE-1, SS-DONE, SS-SIMP, SS-DONESIMP, SS-APPLY, SS-EXACT, SS-RSTEP-SINGLE, SS-DITEM, and SS-APPLYDIS are all trivial: the lifted hitac tactics used in the evaluation rules are identical to the translation. The rules for the *THEN* tacticals (SS-THEN and SS-THENLIST) are also trivial. Take SS-THEN for instance:

$$\frac{\langle \gamma, sstac_1 \ ; \ ALL(sstac_2) \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle}{\langle \gamma, sstac_1 \,; sstac_2 \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle} \quad \text{(SS-THEN)}$$

By the inductive hypothesis, we know that $sstac_1$ and $sstac_2$ have the property that if $\langle \gamma, sstac_1 \rangle \Downarrow_{\mathcal{E}} \langle s_1, g_1 \rangle$ then $\langle \gamma, [\![sstac_1]\!] \rangle \Downarrow^{t}_{\mathcal{E}} \langle s_1, g_1 \rangle$ and if $\langle \gamma, sstac_2 \rangle \Downarrow_{\mathcal{E}} \langle s_2, g_2 \rangle$ then $\langle \gamma, [\![sstac_2]\!] \rangle \Downarrow^{t}_{\mathcal{E}} \langle s_2, g_2 \rangle$. Now, the lifted hitac tactic in the rule SS-THEN is identical to the translation; thus, the property holds. Similar arguments can be used to verify the rules SS-THENLIST, SS-BY, SS-FIRSTLAST-1, SS-FIRSTLAST-2, SS-LASTFIRST-1, SS-LASTFIRST-2, SS-INTRO-1, SS-HAVE-2, SS-REWRITE, and SS-DISCHARGE. The final rules SS-FIRST-1, SS-FIRST-2, SS-LAST-1, and SS-LAST-2 require a little more reasoning, but can be shown using the following technique. Take SS-FIRST-1 as an example:

$$\frac{\langle \gamma, sstac_1 \rangle \Downarrow_{\mathcal{E}} \langle s_1, [\gamma_1, \ldots, \gamma_n] \rangle \qquad \langle \gamma_1, ([first]\ sstac_2) \rangle \Downarrow_{\mathcal{E}} \langle s_2, g \rangle}{\langle \gamma, sstac_1 \ ; \ \textsf{first}\ sstac_2 \rangle \Downarrow_{\mathcal{E}} \langle s_1 \ ; \ (s_2 \otimes id^{n-1}_{\otimes}), g@[\gamma_2, \ldots, \gamma_n] \rangle} \quad \text{(SS-FIRST-1)}$$

By the inductive hypothesis, we know that $\langle \gamma, [\![sstac_1]\!] \rangle \Downarrow^{t}_{\mathcal{E}} \langle s_1, [\gamma_1, \ldots, \gamma_n] \rangle$ and that $\langle \gamma_1, [\![sstac_2]\!] \rangle \Downarrow^{t}_{\mathcal{E}} \langle s_2, g \rangle$. The resulting goal list in SS-FIRST-1 is $g@[\gamma_2, \ldots, \gamma_n]$. The result of the translation is

$$[\![sstac_1]\!] \ ; \ ([first]\ \ [\![sstac_2]\!]) \otimes ID$$

This is evaluated first using the hitac evaluation rule B-TAC-SEQ. The first side of this sequence is evaluated successfully to the list $[\gamma_1, \ldots, \gamma_n]$ by the inductive hypothesis. This list of goals is then passed to the second part of the sequence: $([first]\ \ [\![sstac_2]\!]) \otimes ID$. An application of the rule B-TAC-TENS (with the appropriate split of goals) shows the need to prove that $([first]\ \ [\![sstac_2]\!])$ evaluates with remaining goals g and that ID evaluates with remaining goals $[\gamma_2, \ldots, \gamma_n]$. The first is true by an application of the rule B-TAC-LAB and an instance of the inductive hypothesis; the second is trivial. □

6.6 SEMANTICS FOR SCRIPTS

Recall that eSSence scripts are simply *paragraphs*, which are represented as a pair of lists: a *sstac* list and a *sspara* list. To evaluate scripts, the evaluation relation is

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

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

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

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

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

Figure 6.6: eSSence script evaluation rules

extended to operate on a list of open goals and returns an updated list of remaining goals and a hiproof. The rules are given in Figure 6.6.

The idea is that given a paragraph (a pair of sentence and sub-paragraph lists), each sentence is evaluated 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). The ith paragraph is applied to the ith goal, then the proofs are labelled and glued together (SS-ParaCons). Each sentence is also labeled to make clear the script structure. Sentences and paragraphs are simply labelled with an S or a P. It can be shown that the evaluation relation behaves suitably:

Theorem 23 (Correctness of Evaluation)**.** *If* $(\Gamma \vdash \ \mathsf{P}) \equiv \gamma$ *is a well-formed proposition, and* $\langle \gamma, sstac\rangle \Downarrow_{\mathcal{E}} \langle s, g\rangle$ *then* s *is* ***valid****. That is,* $s \vdash [\gamma] \to g$*.*

Proof. This is an induction on height of the derivation, with appeals to sentence correctness, induction hypothesis, and the relevant hiproof evaluation rules where necessary. As an example, I detail the case for SS-TacCons. We need to prove that if

$$\langle [\gamma], (sstac :: sstacs, ssparas)\rangle \Downarrow_{\mathcal{E}} \langle ([S]\ s_\gamma); s, []\rangle$$

then

$$([S]\ s_\gamma); s \vdash \gamma \to [].$$

Now, looking at the rule:

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

we know that since $\langle [\gamma], sstac\rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, [\gamma']\rangle$ then $s_\gamma \vdash [\gamma] \to [\gamma']$, by correctness of sentence evaluation. Similarly, the induction hypothesis allows us to conclude that

```
1 move ⇒ hAiBiC hAiB hA.
2 apply: hAiBiC ; first by [].
3 by apply hAiB.
```

Listing 6.2: A proof of $(A \to B \to C) \to (A \to B) \to A \to C$

$s \vdash [\gamma'] \to []$. Putting these together with an application of H-SEQ and H-LAB we obtain the result. □

6.7 EXAMPLE.

Recall the proof of $(A \to B \to C) \to (A \to B) \to A \to C$, which is shown again in Listing 6.2 with sentence numbers labelled. 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). A high-level view of the resulting hiproof is shown in Figure 6.7, with the goal context shown alongside. At the level of sentences:

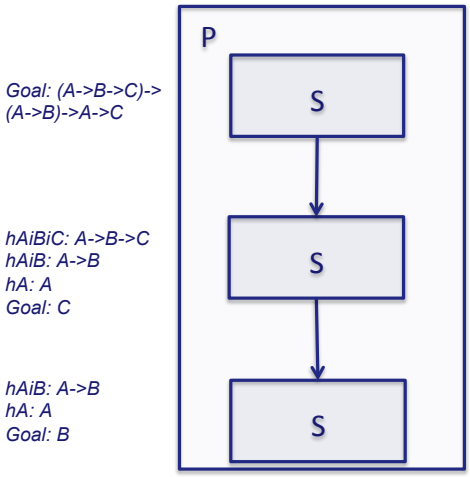

Figure 6.7: High level view of Hiproof

- Sentence 1 is evaluated by first applying the rule SS-INTRO-1, where **move** behaves like an identity then each *iitem* applies the INTRO tactic to generate a context:

  `hAiBiC` : $(A \to B \to C)$

  `hAiB` : $(A \to B)$

  `hA` : A

  ==========

  C

- Sentence 2, whose root is an application of the **first** tactical (where $sstac_1$ is **apply**: hAiBiC and $sstac_2$ is **by** []), is evaluated by SS-First-1 on the new goal. The apply-discharge tactical is then evaluated (SS-ApplyDis) and applies the *ditem* to generate *two subgoals*: A and B in a context with `hAiB` and `hA`. Now, the second part of the **first** tactical — the closing tactical **by** [] — is evaluated and solves the first of these goals.
- The final sentence is used to solve the goal `hAiB` : $A \to B$, `hA` : $A \vdash B$. and proceeds similarly to sentence 2.

Figure 6.8 shows one *view* of the hiproof generated by execution of *sent2*, in context with the rest of the proof. Here the hierarchical structure introduced by each tactical can be seen, as well as the hiding of the proof branch introduced by the **apply** statement.

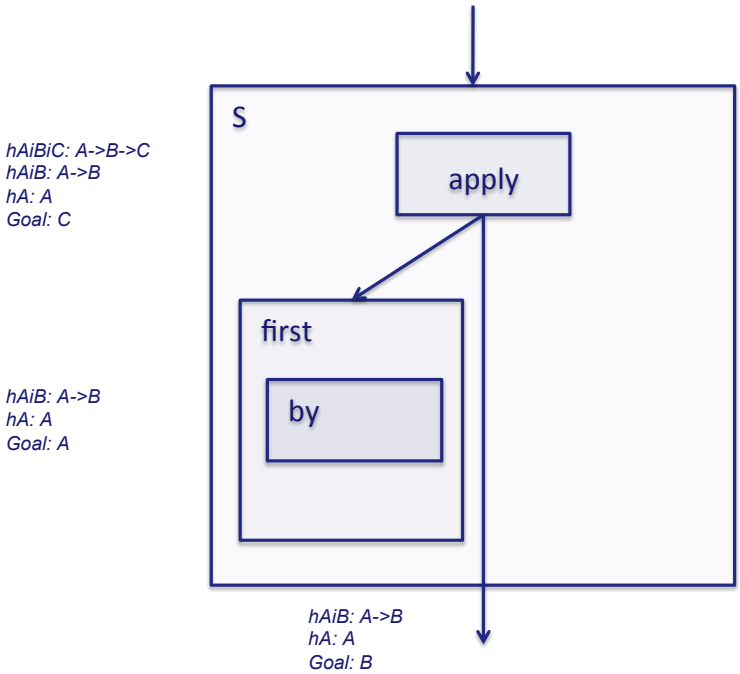

Figure 6.8: The Hiproof for *sent2*

6.8 SUMMARY

In this chapter, I have identified and provided a semantics for a subset of the SSReflect language dealing primarily with proof style. The subset, called eSSence, is introduced by example and given a formal semantics that constructs hiproofs. The hierarchy is used to add structure to the underlying proof and, furthermore, is shown to match the structuring mechanism advocated by Gonthier. The semantics for the language is shown to be correct in the technical sense that it constructs valid hiproofs. In particular, this correctness property is shown by providing a direct translation of the eSSence language to a corresponding Hitac tactic.

Part II

REFACTORING PROOFS

7

IMPROVING THE DESIGN OF EXISTING CODE

7.1 INTRODUCTION

Improving the design of existing code: so runs the subtitle of Martin Fowler's 'bible' of refactoring. This 400 plus-page book catalogues over 70 refactorings — semantics preserving transformations — for object-oriented programs. But what exactly is a refactoring and what is it for?

This chapter introduces refactoring: first in its original incarnation with programming languages, in the next section. Then I motivate and introduce the analogous concept of *proof refactoring* in Section 7.3. Section 7.4 considers the important concept of semantics preservation. I fill in background material by surveying some of the main techniques for programming language refactoring in Section 7.5. Finally, in Section 7.6, I sketch the details of the approaches to proof refactoring described in this thesis, including a breakdown of the structure of this second thesis part.

7.2 INTRODUCING REFACTORING

William Opdyke — whose seminal PhD thesis from University of Illinois in 1992 coins the term refactoring — describes it as:

> A semantics preserving restructuring operation 'that support[s] the design, evolution and reuse [of software]'. (Opdyke, 1992).

```
module SumSquare where

sum_sq :: [Int] -> Int
sum_sq []     = 0
sum_sq (h:t) = h^2 + sum_sq t

main = sum_sq [1..50]
```

Figure 7.1: A simple Haskell program

```
module SumSquare where

sum_sq :: Int -> [Int] -> Int
sum_sq n []     = 0
sum_sq n (h:t) = h^n + sum_sq t n

main = sum_sq 2 [1..50]
```

Figure 7.2: Generalised summation function

I'll try to convey the idea behind refactoring with an example in the Haskell programming language (Peyton Jones et al., 2003). Figure 7.1 shows a very simple Haskell function, written to sum the squares of a list of integers. The `main` function — the entry point to any Haskell program — computes the sum of squares for a list of numbers from 1 to 50 (42,925). Later, I need to compute the sum of cubes and

other powers. Rather than creating separate functions for each power, I realise that I can maintain a *well-designed* development by generalising the `sum_sq` function. As requirements *evolve*, then so must the software.

A refactoring called *generalise function* can help do this, resulting in the program in Figure 7.2. This refactoring takes a subexpression '2' and replaces it with a formal parameter of the appropriate type, thus generalising the function. From this example, the meaning of semantics preservation becomes clear: the *call* to `sum_sq` has had its new parameter instantiated with the original expression, 2; thus, when evaluated, the program still returns 42,925.

Now I have another headache: the function is called `sum_sq`, but that's not quite right: this function can sum any power. Thus, to ensure *maintainability* — imagine the curses of another developer trying to figure out the code — I perform another refactoring: *rename function* (technically, *rename module* is also needed). This refactoring has simpler behaviour, but when choosing a new name, I must ensure that it will not clash with any previously defined names. Having ensured that `sum_pow` is fine, the fully refactored code is shown in Figure 7.3.

```
module SumPower where

sum_pow :: Int -> [Int] ->  Int
sum_pow n []     = 0
sum_pow n (h:t) = h^n + sum_pow t n

main = sum_pow 2 [1..50]
```

Figure 7.3: Generalised summation of powers function

This refactoring illustrates some important elements of refactoring:

- A refactoring may be pervasive — triggering a cascade of changes throughout a software development — but those changes may be routine, such as *rename function*. On the other hand, refactoring may be localised but require sophisticated transformations to code, such as *generalise function*.
- Refactorings are not universally applicable. Typically, *preconditions* are required to ensure that a refactoring will succeed. For example, to use *rename function*, one must ensure that the new name chosen is *fresh*: that it will not clash with any previously used names.
- Refactorings may often have to be chained together or composed. In the example above, the *rename function* refactoring was needed immediately after generalisation to preserve consistency of the development.
- A refactoring should preserve the functionality of a program: it should neither introduce nor remove any behaviours. In Haskell, semantics preservation would mean the identical behaviour of `main` before and after the refactoring.
- Software engineers have been performing refactorings by hand, using search and replace in text editors, for many years but this is an error-prone task for

> even the most basic refactorings. Ideally refactorings are tool-assisted: an algorithm checks safety pre-conditions before making global changes in one go.

Now, over 20 years since Opdyke coined the term, programming language refactoring is an active and challenging research area and a refactoring engine is now seen as a crucial tool as modern software developments regularly reach hundreds of thousands of lines of code. In fact, by analysing four large datasets developed from over 13,000 developers, Murphy-Hill et al. (2009) were able to demonstrate experimentally that refactoring is a very frequent process; interestingly, though, they also discovered that many refactorings are still performed by hand. This suggests that Fowler's catalogue may still be a valuable source, even with automation available.

7.3 WHY REFACTOR PROOF?

The field of interactive theorem proving is maturing rapidly. Recent work on operating system kernel verification has seen the size of the largest development leap pass 500,000 lines of proof (Klein et al., 2009; Bourke et al., 2012) and Gonthier and his team recently announced the completion of their formalisation of the famous Feit-Thompson theorem, which weighs in at 170,000 lines and contains 4,300 theorems (Gonthier et al., 2007; Knies, 2012). The original informal proof was part of the categorisation of finite simple groups in which Aschbacher quipped that

> 'the probability of an error in the proof is one' (Aschbacher, 2005),

which makes a fully verified version of this proof all the more important.

Furthermore, Tom Hales' Flyspeck project to formally prove the famous Kepler's conjecture about sphere packing is in the final stages and may become the largest formal proof yet (Hales, 2006).

With these large projects have come pleas from the trenches for support for *proof refactoring* with Gonthier noting that he had to spend a number of months refactoring his Four Colour Theorem development by hand (Bourke et al., 2012; Gonthier, 2008).

Just as the process of programming is similar to proving, some of the maintenance activities like refactoring are also similar. Renaming lemmas, for example, is similar to renaming functions in programs. Furthermore, many theorem provers implement a small functional programming language for constructing definitions (about which theorems can be proved). Thus, refactorings such as *generalise definition* also have an analogous refactoring for proof developments. However, for proofs the refactoring will also modify any instance of these functions in lemmas that use the definition. To illustrate, Figure 7.4 and Figure 7.5 show a proof development before and after performing the same generalisation refactoring that was performed for Haskell in the previous section. In this case, references to `sum_sq` in the proof have also been changed as well as *rename function* and *rename lemma* being employed.

It is good that I can take inspiration from programming language refactoring research, but there are also opportunities for refactoring formal proofs that have no counterpart in programming languages. One example is a refactoring that transforms a backwards style proof to a forwards style proof.

Finally, while the 'types' of refactoring that I would like to perform are clear, what it means for a proof refactoring to be semantics preserving is not quite so obvious.

```
fun sum_sq :: "int list ⇒ int"
  where
  "sum_sq [] = 0" |
  "sum_sq (x#xs) = x^2 + sum_sq xs"

lemma sum_sq_nonneg:
  fixes xs :: "int list"
  shows "sum_sq xs ⩾ 0"
proof (induct xs)
  show "0 ⩽ sum_sq []" by simp
next
  fix a xs
  assume IH: "0 ⩽ sum_sq xs"
  show IC: "0 ⩽ sum_sq (a # xs)"
  proof (simp)
    have "a^2 ⩾ 0" by simp
    from this and IH
    show "0 ⩽ a^2 + sum_sq xs"
     by (rule add_nonneg_nonneg)
  qed
qed
```

Figure 7.4: An Isabelle definition and lemma

```
fun sum_pow :: "int list ⇒ nat ⇒ int"
  where
  "sum_pow [] n = 0" |
  "sum_pow (x#xs) n = x^n + sum_pow xs n"

lemma sum_pow_two_nonneg:
  fixes xs :: "int list"
  shows "sum_pow xs 2 ⩾ 0"
proof (induct xs)
  show "0 ⩽ sum_pow [] 2" by simp
next
  fix a xs
  assume IH: "0 ⩽ sum_pow xs 2"
  show IC: "0 ⩽ sum_pow (a # xs) 2"
  proof (simp)
    have "a^2 ⩾ 0" by simp
    from this and IH
    show "0 ⩽ a^2 + sum_pow xs 2"
     by (rule add_nonneg_nonneg)
  qed
qed
```

Figure 7.5: Refactored Isabelle theory

There is no longer a `main` function whose behaviour must be preserved. The next section discusses the possibilities for semantics preservation.

7.4 SEMANTICS PRESERVATION

In programming languages, semantics preservation for refactoring is often seen informally: given a finite set of test inputs, a refactoring has succeeded if you get the same results before and after the refactoring. This doesn't guarantee that there is no input that produces divergent behaviour.

Arguing more formally about correctness of refactoring — for all possible inputs — requires a formal semantics to be assigned to the languages, which is often not possible or loosely defined. Even with a formally defined refactoring, often the translation to an implementation (manipulating the Abstract Syntax Tree (AST)) is not direct. More recent work has seen more formal approaches to refactoring; however, with tools in the wild, a thorough testing procedure by developers is important for a guarantee of behaviour preservation. The various approaches to refactoring are surveyed in the next section.

So what about *proof refactoring*? The equivalent to an *entry point* or *main* function for a proof development would be to choose a 'final' theorem that should still be proved after refactoring. Often though, there is not one theorem that a proof engineer would like to ensure is preserved, but many, or even all. A possible notion for semantics preservation for proof refactoring would be that all lemmas proved before the refactoring are proved afterwards. At first sight, then, with proofs, testing gives a lot more assurance. By *testing*, I mean replaying the proof script. If I test a proof and

it succeeds before and afterwards, then the refactoring has been performed correctly. But, there are two complications:

- Proof checking can be a time-consuming process, so it is beneficial to minimise the proofs that need checked. Furthermore, changes made by a refactoring could cause a tactic to loop infinitely.
- Rather than being happy with the proof of a lemma still succeeding, the defining characterisation of the lemma is not the proof, but the statement: what is actually proved. Thus, what I actually wish to ensure is that the same *statements* are proved before and after. This then, needs manual inspection or even reasoning about equivalence of the two statements. Take *generalise definition* for example: ostensibly the statements in Figures 7.4 and 7.5 are different, but semantically they are the same.

Thus, 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 by Pons et al. (1998).

7.5 A SURVEY OF REFACTORING

Over the past 20 years, many approaches to programming language refactoring have been studied. In this section, I briefly survey the most relevant research to the work presented in this thesis. I reserve a direct comparison with my approach until Chapter 12.

7.5.1 *Informal refactoring presentations*

Refactoring has its origins in object-oriented programming, where developers often need to 'redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptions and extensions' (Mens and Tourwe, 2004). Opdyke (1992) specified 26 'low-level' C++ refactorings, such as:

- **Create a member variable/function/class.** These refactorings introduce a new variable or function to a class or create a new, empty class. Although this appears to be a trivial refactoring, it provides a structured approach to adding to a development.
- **Delete an unused member variable/function/class.** These refactorings will remove a variable, function or class provided that it is not referenced anywhere.
- **Rename a member.** This class of refactorings are used to rename variables, functions and classes, consistently renaming any calls to these within the code.
- **Move a member variable.** These refactorings are used to redistribute a variable to a sub- or super-class.

These refactorings are specified as functions with:

1. A set of arguments. For example, for *move a member variable* the arguments are V: the variable to move; and, C: the class to move to.

2. A list of preconditions. These preconditions are stated using first-order logical connectives within a domain of primitive predicates and functions. For example, a precondition to the *move a member variable* refactoring is:

   $$\forall\, member \in C.locallyDefinedMemberVariables.\ member.name \neq V.name$$

   or, in English, the variable name has not already been defined.

3. An algorithm for the implementation of the refactoring. This is either in the form of step-by-step manipulations or a natural language description.

Opdyke also specifies three 'high-level' refactorings, defined as compositions of the 'low-level' refactorings, showing how the preconditions for each low-level step are satisfied. This idea of composition is an important idea in research into refactoring: in order to show behaviour preservation of a complex refactoring, it is sufficient to describe it as a sequence of simple refactorings, where behaviour preservation is more obvious.

Fowler (1999) takes a similar approach to the specification of refactorings. 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. A simple example is the *encapsulate field* refactoring. This refactoring will make a public variable private, creating accessor functions. A variable (field in OOP parlance):

```
1 public String name;
```

will be converted to:

```
1 private String name;
2 public String getName() {return name;}
3 public void setName(String n) {name = n;}
```

The 'mechanics' of this refactoring are the following steps:

1. Create *get* and *set* methods for the field.

2. Find all methods outside the current class which reference the field. Replace these with a call to the *get* method.

3. Find all methods which update the field. Replace these with a call to the *set* method.

4. Compile and test after each change.

5. Declare the field as private.

6. Compile and test.

In his thesis, Roberts (1999) discusses how to implement a tool that will perform a refactoring automatically. The *Refactoring Browser* was built to refactor Smalltalk

programs and was the first automated refactoring tool. Roberts also provides a different viewpoint on the specification of refactorings, giving a definition of a refactoring as a triple, (pre, T, P), where pre and T are the preconditions and transformation as suggested by Opdyke. The function P, however, is used to transform assertions (in First Order Predicate Logic) about the program when T performs the transformation. The motivation behind introducing post-conditions is to allow dependencies between refactorings to be calculated and derive preconditions for composite refactorings.

As a concrete example, consider a refactoring *rename class* which takes two arguments *class* and *newClass*: the old and new class names. A precondition of this operation is:

$$\mathrm{isClass}(class) \wedge \neg \mathrm{isClass}(newClass)$$

A postcondition of this operation is the new assertion:

$$\mathrm{isClass}? = \mathrm{isClass}[class/false][newClass/true]$$

where isClass is modified to reflect the new nature of the program.

7.5.2 *Formal programming language refactoring*

Garrido and Meseguer (2006) notes a disadvantage of the semi-formal approach to refactoring, where transformations are specified using natural language: it is difficult to provide a formal proof of the behaviour preservation of the refactoring. Furthermore, in many cases, even if an adequate specification is provided, the implementation may still be incorrect.

By using executable equational semantics for the Java language, the authors provide a formal specification for the refactoring in Maude, a rewriting system, which can be directly executed to refactor a Java program (Clavel et al., 1999). The authors also provide proofs of behaviour preservation of a program after refactoring. The disadvantage with this approach is that the specifications of refactorings are difficult to understand and the proofs even more so.

Cornélio et al. (2002) specify refactorings for a language implementing a subset of Java, called ROOL (Borba and Sampaio, 2000). A set of basic algebraic laws, expressing equivalences between objects, is provided for this language. A simple law states that two classes, which are identical except one has an additional attribute, are equivalent as long as that attribute is not used in the class. The authors specify refactorings such as *extract method, move method, encapsulate field* and *extract superclass* as derived algebraic laws, with suitable preconditions. Correctness is shown for an example refactoring by deriving it from the basic laws.

As a way of reconciling correctness and understanding, Mens et al. (2005) argues that a formalism for refactoring must satisfy four criteria:

1. Transparency: it must be as close as possible to the underlying implementation of the refactoring.
2. Conciseness: simple refactorings should be stated in one step, whereas more complicated should be no more than a few.
3. Elegance: it should be clear from inspection what the refactoring is doing.

4. Expressiveness: it should be able to express most of the core concepts in the language under refactoring.

To this end, Mens et al. (2005) considered a graphical formalism using an embedding approach with *graph productions*. Each refactoring would be specified as sequential and/or iterated application of productions. A program would be represented as a graph, with typed edges describing the structure. The authors considered a weakened model of behaviour preservation. They consider three types of preservation:

- Access preservation: each method accesses *at least* the same variables after the refactoring.
- Update preservation: each method performs *at least* the same variable updates after the refactoring.
- Call preservation: each method performs *at least* the same method calls after the refactoring.

Eetvelde and Janssens (2003) introduce a hierarchical structure to these program graphs based on four levels of abstraction. At the highest level of abstraction, the only information contained in the graph is names and types. At the next level, information about where each variable and method is implemented is supplied. At the third level, information about method calls, and variable update and access is stored. Finally, control flow information is added at level four. Refinements between these levels are described by graph productions. The benefit of this approach is that different components of a refactoring can be described on different levels, reducing complexity of specifications. For example, to perform the *Pull-up Method* refactoring only requires information from the second level; however, to check the preconditions requires level three information.

Some of the most recent work on refactorings has been a project led by Simon Thompson to develop a refactoring tool for the Haskell programming language (Peyton Jones et al., 2003). The project aimed to investigate refactoring from a functional programming perspective, where immutability of data offers greater scope for change, and to develop a prototype tool with a modest catalogue of available refactorings. In Huiqing Li's PhD thesis, a number of refactorings are identified and classified into three broad categories: structural, modular and data-oriented refactorings (Li, 2006). A simple example of each category is given:

GENERALISE A DEFINITION. This structural transformation allows the user to identify a subexpression on the right hand side of a function to be passed as a new formal parameter. Crucially, any calls to this function should have the original subexpression in this parameter position.

MOVE A DEFINITION This modular transformation allows the user to move a definition from one module to another, ensuring that, for any calls to the function, it will still be in scope in its new position.

CONCRETE TO ABSTRACT DATA TYPE. This data-oriented transformation takes a user-defined datatype and constructs an abstract data type, providing constructor and discriminator functions for it.

Li also considers the problem of *appearance preservation*: as a refactoring is essentially a source-to-source change, it should preserve as much user layout style and commenting information as possible. The solution, implemented in their tool HaRe, is to use information from both the Abstract Syntax Tree (AST) — which doesn't contain syntactic information — and the token stream — which does — to perform the refactoring (Li et al., 2005). The transformation is performed on the AST, but it is merged with the token stream, using location information such as line and column numbers, to provide new source code with the appropriate layout information still available.

Li and Thompson (2005) discuss a formal specification of refactorings based on an abstract representation of a program before and after application of the refactoring and provide a proof that the semantics of the program are preserved during the refactoring by providing a step-by-step transformation of the program and showing that each atomic transformation is safe. The formal representation used is the λ-calculus extended with a *letrec* construct. The semantics of the program is expressed by a modified call-by-need reduction relation and semantics preservation is given by reduction of both programs to the same value. This formal verification is a powerful concept as often refactorings are presented as algorithms to follow, but it is not clear that they are correct for every scenario. Sultana and Thompson (2008) go one step further and mechanically prove specifications in Isabelle/HOL. While this takes a promising step towards correctness guarantees, there is a discontinuity in this approach to formalisation: the formal specifications are completely separate from the implementation of the refactoring and are verified against a simplified version of the language and its semantics.

7.5.3 *Refactoring tools*

Many tools have been developed to automate the refactoring process. The vast majority of tools are for object-oriented programming languages and Java in particular.

The Smalltalk refactoring browser was the first tool designed to automate the refactoring process (Roberts et al., 1997). It was built by Roberts as part of his thesis work. The browser itself supports many of the refactorings described by Opdyke. The tool is still available and is distributed by Refactory Inc (2012).

The Haskell Refactorer (HaRe), developed by Li et al. (2005), is implemented in Haskell and available for developers using Emacs and Vim. This choice was motivated by a study of the most common editors for Haskell developers. The tool is built upon a frontend, for lexing and parsing, and the *Strafunski* library for abstract syntax tree traversals (Lämmel and Visser, 2003). The individual refactorings are then written using these traversals. The available refactorings are selected from a drop-down menu in the interface and the user may be prompted to enter additional information, such as new names. The authors provide an API to allow user development of refactorings.

The Java language has the largest number of tools for automated refactoring. Many of these tools are built into IDEs such as Eclipse (2012) and NetBeans (2012). The catalogue of refactorings in Netbeans for instance contains some of the standard refactorings such as *rename a class/method*, *delete a method*, and *encapsulate field*. There are also some more complicated refactorings such as *extract method*. Eclipse also supports

these refactorings, and provides APIs to allow a programmer to develop their own refactoring. There is an issue with transparency here: it is difficult to know whether these refactorings are correct implementations. In Ettinger and Verbaere (2005), there are over 20 bugs listed for Eclipse refactorings. The refactorings are performed on the source code by using the dependency information stored in the program database and information from the abstract syntax tree.

Mens et al. (2003) suggest that performing a refactoring is really only the final stage in a three-part process:

1. Detect when code needs refactored: the so-called *bad smells*.
2. Identify which refactorings can be performed and where.
3. Perform the refactoring.

To demonstrate this, they developed a prototype tool which builds on the Refactoring Browser of Roberts et al. (1997). The tool provides a small number of queries, which can be run on the code-base, such as: *'Are there duplicated methods?'* and *'Is this a large class?'*. These queries, when run, identify any instances where a refactoring could be performed and allow the user to perform it.

7.5.4 *Generic and language independent refactorings*

In Verbaere et al. (2006) a domain-specific programming language called JunGL, designed to enable a programmer to write their own refactorings, is described. The language combines ML-style functional programming with a logic query language like Datalog. The language is generic in the sense that implementation for a new programming language requires a parser for the new language, alongside a so-called type graph for the language, which constrains when the graph represents a well-formed program in the language. Then, refactorings can be written for the new programming language in JunGL. Lämmel (2002) take a different approach to genericity, and describes a prototype framework for generic refactoring. The approach is based on generic programming to traverse ASTs. Generic algorithms are provided to perform simple analysis and atomic transformations. Then, an abstraction interface must be provided for each language to deal with the specific components of each language.

7.5.5 *Refactoring in other paradigms*

The Z specification language is used to formally describe software systems and is based on a typed set theory. The language facilitates models of the computing system to be developed and proofs of certain desirable properties of these systems can be provided. Stepney et al. (2002) suggest refactorings of these 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. A similar effect will happen in proof scripts: if a definition is changed, all proofs using that definition will be affected. The paper lists some example refactorings, which have analogues in a proof script:

- Turn a common proof step into a lemma;
- Move a common proof step to before a branch point;
- Merge state components.

Furthermore, the authors also consider 'benefactorings' as more general refactorings which actually change the semantics of the program. A typical example, also of interest to theorem proving, is *change a type*. The Rodin Toolkit, for the Event-B specification language, contains a basic refactoring tool which can perform safe renaming (Butler and Hallerstede, 2007). Sunyé et al. (2001) focus on refactoring UML models using a graphical approach and show the specification and correctness of a *move attribute* refactoring. Boger et al. (2002) implemented a prototype tool for automatically performing UML refactorings.

7.6 APPROACH

Taking inspiration from research into programming language refactoring and mindful of the importance of semantics preservation for formal proofs, I identified a number of aims that this thesis would address:

- Refactorings should be defined formally, along with an appropriately related statement of correctness.
- The refactoring specifications should be close to an implementation, to provide more assurance of correctness.
- The theorem proving community is diverse; thus, an approach to proof refactoring that is generic would be useful.

The following four chapters present the results of my research into proof refactoring. The work has had three separate stages:

1. Chapter 8 further introduces proof refactoring with a small selection of refactorings in the style of Fowler (1999). The chapter provides an informal description, motivation and step-by-step recipe for performing seven of the proof refactorings that I formalise.
2. Chapters 9 and 10 form an investigation into providing formal refactoring specifications for Hiscript. These chapters specify and prove correct over 20 refactorings for Hiscript, including those described in the previous chapter, by providing transformation rules that act on the syntax of the language. Appendix B provides brief introduction to the Hiscript framework.
3. Finally, Chapter 11 describes joint work with Dominik Dietrich into a formal, generic framework for refactoring proofs. Our framework allows for declarative specification of refactorings using graph rewrite rules that act on an abstracted representation of a proof language in a graph meta-model.

The thesis concludes with a discussion of further work in Part 3, Chapter 12.

8

A CATALOGUE OF PROOF REFACTORINGS

8.1 INTRODUCTION

This chapter further introduces proof refactoring by providing a modest collection of informal refactoring descriptions. There is a precedent for such a 'catalogue'. In the domain of programming language refactoring, Fowler's book 'Refactoring: improving the design of existing code' includes a catalogue of over 70 refactorings and is still considered the 'bible' of object-oriented program refactoring (Fowler, 1999).

Each proof refactoring in this chapter is named and has a description of its behaviour. To build context, I also provide some motivation for applying this refactoring. Then, I provide (usually) one example of the refactoring in action for a particular proof language and describe the *mechanics* of the transformation; that is, I provide a pseudo-algorithm that could be followed to perform the refactoring.

CHAPTER MAP This chapter contains seven informal refactorings:

- *Rename item*, described in Section 8.2.
- *Copy item*, described in Section 8.3.
- *Move item within theory*, described in Section 8.4.
- *Flatten a subproof*, described in Section 8.5.
- *Fold a tactic*, described in Section 8.6.
- *Unfold a tactic*, described in Section 8.7.
- *Generalise a tactic*, described in Section 8.8.

I then give briefly summarise in Section 8.9.

CONTRIBUTIONS While this chapter is mainly introductory, it contributes towards the body of knowledge of refactoring by providing a small set of refactorings for proof documents and providing motivation and an informal recipe for performing that refactoring.

8.2 RENAME AN ITEM

DESCRIPTION The classic structural programming language refactoring is a renaming: a simple action but with complex preconditions and global consequences. This

section describes how to change the name of an object in a theory: a lemma, a tactic, a definition etc.

The problem with naming items is that good names are hard to think of, especially in the heat of the moment. Choosing good names requires practice. If a name is found to obscure its purpose, it can be seen as prohibitive to put the energy into changing it, especially if it is used many times throughout a development and without support for automating part of the task. Describing renaming as a refactoring gives a structured technique for renaming items without the worry that a particular reference has been missed.

Importantly, the new name that has been chosen must be *fresh* to ensure that it does not cause any clashes with other named items. Freshness is a concept that is different for every proof language: languages have different scoping rules and different namespaces. I do not attempt to give precise rules for freshness, but instead discuss name freshness in terms of how it would affect behaviour of a modified theory. There are two ways in which a proposed new name can conflict with existing names:

- The proposed name could already be defined in the appropriate namespace. Thus, depending on whether the language allows shadowing[1] or not, the renamed item would fail to evaluate or would change the meaning of any references to the shadowed name later in the theory.
- The proposed name will be defined at some point later in the theory where, importantly, the item to be renamed is still in scope. Thus, depending on whether shadowing is allowed, evaluation would fail or behaviour would be changed.

MOTIVATION Good names are evocative and can help shed light on a concept; at the same time, however, badly chosen names are like the brain teaser where you have to read the colours of a group of words correctly (Figure 8.1). One part of your brain instinctively reads the word which doesn't match the colour. If you do not choose good names for facts, tactics, lemmas then they become hard to find, harder to reuse, harder to comprehend in proofs.

Figure 8.1: Say the colour, not the word

As anecdotal, expert evidence for the importance of a renaming refactoring, during conversations with Georges Gonthier, I learnt that he believes choosing the correct name is an art-form; having a consistent policy for names is important to reduce

1 Shadowing is where two variables, one in an inner and one in an outer scope, have the same name. The variable in the outer scope is said to be shadowed.

thinking time; furthermore, a good name should neither be too short nor too long Gonthier (2012).

RECIPE

- ◇ Create a new theory item with the new name.
- ◇ Copy the body of the old theory item to the new one. Be careful to make any changes for recursive calls in the body as they must now refer to the new theory item.
- ◇ Check your proofs still 'replay'; that is, they are evaluated successfully by the proof checker. This ensures that the new name you have chosen is a fresh one.
- ◇ Find all references to the old item and replace them with a reference to the new one. Replay the proof after each change.
- ◇ Delete the old item.
- ◇ Replay proofs a final time to ensure that all references to the old item have been removed.

EXAMPLE The theory in Listing 8.1 contains a proof of commutativity of set intersection. Any further development might prove theorems about commutativity of other binary operators. Therefore, it makes sense to perform a renaming of *comm* to *intersection_comm*.

8.3 COPY AN ITEM

DESCRIPTION This refactoring takes a previously defined theory item (a lemma, a tactic, a definition etc.) and makes a copy of it, with a fresh name supplied by the user. No changes are made to any references to the copied item. If one is copying a recursive definition (or a recursive tactic), changes will be needed to the body of the definition to be consistent.

MOTIVATION This refactoring provides an easy way to experiment with variations on a proof of a lemma, for example, without actually changing the 'live' one.

RECIPE

- ◇ Ensure that the name for the copied item is *fresh*.
- ◇ Copy and paste the whole item just above the current one.
- ◇ Change the name to the new one and ensure to make any recursive calls consistent.
- ◇ Check your proofs still replay.

```
theory set
begin

 ...

public hitac intro := ⊆—def | ∩—def | id

public lemma comm: A ∩ B ⊆ B ∩ A
proof( intro )
show x ∈ A ∩ B ⊢ x ∈ B ∩ A
    proof( intro )
      show x ∈ A ∩ B ⊢ x ∈ B
        by ∩—elim ; ax
      show x ∈ A ∩ B ⊢ x ∈ A
        by ∩—elim ; ax
    qed
qed

 ...

end
```

Listing 8.1: The name *comm* is too generic and can cause confusion

```
theory test
imports main
begin
 …
fun sum_sq :: "int  list  ⇒ int" where
"sum_sq [] = 0" |
"sum_sq (x#xs) = x^2 + sum_sq xs"
 …
end
```

Listing 8.2: A snippet of an Isar proof with a definition of summing the squares of a list

```
theory test
imports main
begin
 …
fun mult_sq :: "int  list  ⇒ int" where
"mult_sq [] = 0" |
"mult_sq (x#xs) = x^2 + mult_sq xs"

fun sum_sq :: "int  list  ⇒ int" where
"sum_sq [] = 0" |
"sum_sq (x#xs) = x^2 + sum_sq xs"
 …
end
```

Listing 8.3: Isar proof with copied definition ready for a semantic change

EXAMPLE It is possible to modify the function definition 'sum_sq' to multiply the squares of a list. As a first step to such a semantic change, one could copy the definition and rename it to 'mult_sq'. If the initial theory looked like Listing 8.2, then the refactored theory would look like Listing 8.3. Note that the resulting definition doesn't quite match its name: this is for the forthcoming semantic change to sort out.

8.4 MOVE ITEM

DESCRIPTION The typical evaluation model of theories in proof assistants is that theory items are evaluated step-by-step: building up a proof environment. This linear evaluation model ensures items are defined before they are used. This means that if an item X depends on item Y, then it must come after Y in the linear script. This dependency information can be used to move theory items up or down in a theory so long as they don't interfere with this dependency structure.

The best way to perform such a transformation by hand is to make stepwise changes: swapping the position of two adjacent items until the item you would like

moved is in the correct place (if that is possible). Swapping two objects is simpler as it only requires local dependency information: given a theory snippet:

```
lemma lem_1: γ_1
proof
  ...
qed

lemma lem_2: γ_2
proof
  ...
qed
```

lem_1 can be moved below lem_2 only if it is not used in the proof of lem_2. For Hiscript theories, introduced in Chapter 5, there is a precise notion of what it means for a lemma or a tactic to be used in a proof. In fact, this property is used to prove correctness of this refactoring in Chapter 9. However, for many other systems, such as Isabelle, powerful tactics like the simplifier can use lemmas with references in the text. In fact, in the Coq system, Pons showed that two independent lemmas — with no references even in the full proof term — could still have a dependency (Pons et al., 1998).

Interestingly, this refactoring has been used in large-scale development and during the seL4.verified project, two tools were used to help perform it: *Gravity* — developed as part of the Verisoft formalisation (Alkassar et al., 2009) — for analysing the minimal dependencies and *Levity* for actually performing the moving (Klein et al., 2009; Bourke et al., 2012).

MOTIVATION There are a few good reasons for restructuring theories. When managing a large proof development, it is good practice to group related lemmas together; in particular, if a group of lemmas relate to a definition, placing them near to it creates a nice context. Additionally — and this is especially important for large developments — placing a theory item as far up the dependency graph as possible may reduce the amount of unneeded lemmas and tactics available to a simplifier tactic, for example, thus speeding up the evaluation.

RECIPE To perform this refactoring, follow a step-by-step approach: moving the theory item up or down one step at a time until you either reach the desired location or a swap fails. If it fails, you have a choice of where to place it: anywhere between the current location and the original location. Individual swaps can be performed by:

- ◇ Check if the item you are about to move up/down has any explicit dependency on the other item.
- ◇ If it hasn't, simply copy and paste to the new location.
- ◇ Attempt to replay your proofs. If they succeed, you can continue to move the item.

```
lemma conj_comm : P ∧ Q ⇒ Q ∧ P
proof (impl)
  show P ∧ Q ⊢ Q ∧ P
  proof
    have q: P ∧ Q ⊢ Q by conjE ; ax
    have p: P ∧ Q ⊢ P by conjE ; ax
    from q p show P ∧ Q ⊢ Q ∧ P by conjI
  qed
qed
```

Listing 8.4: Here the inner proof block is not necessary

```
lemma conj_comm : P ∧ Q ⇒ Q ∧ P
proof (impl)
  have q: P ∧ Q ⊢ Q by conjE ; ax
  have p: P ∧ Q ⊢ P by conjE ; ax
  from q p show P ∧ Q ⊢ Q ∧ P by conjI
qed
```

Listing 8.5: Removing the proof block preserves evaluation behaviour

8.5 FLATTEN A SUBPROOF

It is possible to take a subproof block i.e. a **proof** ... **qed** and flatten it. That is, move it up a level in the proof hierarchy. To illustrate, consider the forward proof block in Listing 8.4; here the proof block that solves the goal $P \wedge Q \vdash Q \wedge P$ (highlighted) could be flattened, resulting in the proof shown in Listing 8.5.

The reason this refactoring can be performed easily in this instance is because the statements in that proof block contain only one goal-solving step and that step solves the same goal as the outer **show** statement.

This refactoring is more widely applicable, though, and it can flatten almost all proof blocks. The basic idea is to move all statements up to the parent proof block. Schematically, if a proof block looks like Listing 8.6 to begin with then it will look like Listing 8.7 after the refactoring.

```
proof
...

  show n: γ
  proof(t)
    stmt_1
     ...
    stmt_n
  qed

...
qed
```

Listing 8.6: Nested proof block

```
proof
...


  apply t'
  stmt_1
   ...
  stmt_n


...
qed
```

Listing 8.7: Flattened proof block

Note that any proof block introduction tactic t must be transformed into an apply statement. However, apply statements in Hiscript operate on the whole proof context (the full list of remaining goals) whereas t itself just operated on a single goal γ. This means that t needs to be transformed to behave as before on γ — the first goal in the proof context — but do nothing to the other goals. For Hiscript, this is straightforward with the identity tactic *id*:

$$t' := t \otimes id \otimes \ldots \otimes id$$

where the number of identities match the number of extra goals in the proof context. For other languages, such as Isar, a change may not be required: many tactics only operate on the first subgoal anyway; however, a tactic like Isabelle's *auto* — which applies powerful simplification procedures to all goals — will need its behaviour restricted to the first goal.

Now the statements need to be flattened. Similarly to proof block introduction tactics, procedural steps need to be transformed as the scope of the tactic supplied by an apply statement is now the whole proof context, not the nested proof context. Thus, one can again tensor together the appropriate number of identity tactics. Conveniently, the rest of the statements in a proof block can be preserved without changes, as long as any names introduced in the previously nested proof block (by have, for example) will not clash with names further down its parent.

MOTIVATION With too many nested proof blocks, it becomes difficult to keep track of a proof with a lot of space being taken up by indentation and qeds. This refactoring helps with this problem by providing a way to make structured changes to the degree of nesting.

RECIPE

- ◇ If there is a proof block introduction tactic, calculate how many goals are in the context at this point and modify the introduction tactic appropriately: place it above the block you wish to flatten as an apply statement.
- ◇ Delete the proof block delimiters and attached show statement.

```
lemma preflat: P ∧ Q ⇒ Q ∧ P
proof( intros )
  show q: P ∧ Q ⊢ Q
  proof(conjE)
    show P, Q ⊢ Q
      by assumption
  qed
  show p: Q ∧ P ⊢ P
    by conjE ; assumption
qed
```

Listing 8.8: The proof of $P \wedge Q \vdash Q$ can be flattened

```
lemma flat: P ∧ Q ⇒ Q ∧ P
proof( intros )
  apply (conjE ⊗ id)
  show P,Q ⊢ Q
    by assumption
  show p: Q ∧ P ⊢ P
    by conjE ; assumption
qed
```

Listing 8.9: The proof of $P \wedge Q \vdash Q$ after flattening

- ◇ Then, for each statement: if it is a procedural step, append the appropriate identity tactics.

EXAMPLE Consider the contrived proof in Listing 8.8. The nested proof block solving the goal $P \wedge Q \vdash Q$ is unnecessary and can be flattened. The resulting proof is shown in Listing 8.9. Note that one identity tactic is needed to 'skip' the final goal in the context.

8.6 FOLD A TACTIC

DESCRIPTION The *fold tactic* refactoring has a simple description: it allows one to extract a tactic term, give it a name, and define it as a tactic in its own right. Tactics generally occur in proofs, for example:

$$\textbf{by}\ t_1\ ;\ (t_2\ \otimes\ t_3)\ ;\ t_4$$

but also occur in other tactic definitions:

$$\textbf{tac}\ \mathit{mytac}(X, Y)\ :=\ t\ ;\ X\ \otimes\ Y.$$

Of course, the whole term does not need to be folded: sub-tactics can also be extracted (just t_1 or $t_2 \otimes t_3$, for example). Once an appropriate tactic has been selected, a *fresh* name must be chosen and a tactic can be created above the theory item that

```
lemma conj_comm: P & Q ⇒ Q & P
proof(impl)
  have q :  P & Q ⊢ Q
    by conjE ; assumption
  have p : P & Q ⊢ P
  show P & Q ⊢ Q & P
    by conjI ; lem q ⊗ lem p
qed
```

Listing 8.10: A proof from which I wish to extract a tactic (highlighted)

```
tac newtac(X,Y) := conjI ; X ⊗ Y

lemma conj_comm: P & Q ⇒ Q & P
proof(impl)
  have q :  P & Q ⊢ Q
    by conjE ; assumption
  have p : P & Q ⊢ P
  show P & Q ⊢ Q & P
    by newtac(lem q,lem p)
qed
```

Listing 8.11: The extracted tactic along with the two introduced parameters

contains the selected term. The refactoring could then be complete, but there are also two options to continue this transformation:

1. Replace the selected tactic expression with a reference to the new definition.
2. Replace the selected tactic and search the theory for more instances of the given term and replace them all.

Tactic folding also has a subtle complexity, though. Imagine extracting the highlighted tactic term in the proof shown in Listing 8.10. Two of the lemmas that it uses (q and p) are actually defined in the proof. Instead of disallowing this refactoring, I take the approach that tactic variables can be introduced and instantiate them at call sites. Thus, refactoring this lemma would lead to a proof document as shown in Listing 8.11.

MOTIVATION Often, during the course of a proof development you will write tactics that you will use regularly or tactics designed to perform a certain action — like rewriting a term to a particular form — and these are prime candidates to be given an identity of their own. Refactoring these out as named tactics will simultaneously reduce the size of a development and increase the readability as well as enabling a potential semantics change to that tactic in the future.

RECIPE

```
...
tac REPEAT(X) := X ; ALL(REPEAT(X)) | ⟨⟩

tac intros := REPEAT(impl | conjl | alll | notl)
...
```

Listing 8.12: The definition of an *intros* tactic

- ◇ First ensure that the selected tactic term is well-formed in the current proof environment. It may not be if you only select one side of a tensor, for example.
- ◇ Then ensure that the new name you wish the tactic to have is fresh.
- ◇ If the tactic term you are attempting to extract is part of a tactic definition (a **tac** or **hitac** statement) then:
  - Any tactic variables in the selected term need to be identified.
  - You need to ensure there are no recursive calls to the parent tactic. If there were, the new tactic would fail to evaluate since the new tactic would have references to a tactic not yet defined (since the new tactic necessarily comes before the original).
- ◇ If the tactic term you are extracting is part of a proof, instead: check if it has references to locally defined tactics and lemmas as these must be replaced by new tactic variables.
- ◇ Next, copy the tactic and create it as a tactic definition with the chosen name.
- ◇ Replace instances of the now out-of-scope references with tactic variables. Remember that multiple references to the same item should be given the same tactic variable. Also keep a note of the mappings for later.
- ◇ Check now that your proofs still compile.
- ◇ Now, replace the tactic term you have just extracted with a call to the definition, being careful to map the local references to the appropriate parameters.
- ◇ Check your proofs still compile.
- ◇ Optionally, repeat the process for any other instances of the tactic you have just extracted. Check, after every instance, that the proofs still compile.

EXAMPLE A first example of this refactoring is shown in Listings 8.10 and 8.11. However, I also show an example where a tactic definition is also refactored. Consider the tactic definitions in Listing 8.12. It would be nice to be able to only apply a single introduction rule at a time, for more fine control. This could be achieved by folding the highlighted tactic term, calling it *intro* as shown in Listing 8.13.

8.7 UNFOLD A TACTIC

The *unfold tactic* refactoring has the opposite behaviour to the previous refactoring.

```
...
tac REPEAT(X) := X ; ALL(REPEAT(X)) | ⟨⟩

tac intro := impl | conjl | orl | alll | notl

tac intros := REPEAT(intro)
...
```

Listing 8.13: The result of extracting an *intro* tactic

DESCRIPTION The unfold tactic refactoring simply replaces a call to a defined tactic with its body, suitably substituted with variables. As with the fold tactic refactoring, there are a few options for scope of this refactoring. It can be:

- A single unfolding of a single call to a tactic, keeping the original definition.
- An unfolding of all calls to that tactic whilst keeping the original definition.
- Unfolding all calls to a tactic and deleting the definition.

The third option can be seen as exhaustive application of the first option and then an instance of a *delete unused item* refactoring. Each single unfolding would be an instance of a *replace equivalent tactic* refactoring. Again, there is a slight complexity with recursive tactics. Imagine trying to exhaustively unfold a recursive tactic and the problem becomes apparent. This means that if you wish to unfold all calls and delete the definition, the tactic in question cannot be recursive.

MOTIVATION Perhaps a tactic is not so common after all or perhaps it is very specific to a proof: that is, the body of the tactic definition is understandable only in the context of the particular proof. These are two of the reasons for applying this refactoring. While it is important to create abstractions (i.e. folding tactics), one needs to toe a delicate line between too few and too many. This refactoring helps with this particular balance.

RECIPE To perform the basic unfolding, follow these steps:

◇ Create a mapping from tactic variables in the definition to the parameters passed in the tactic call.

◇ Copy the body of the tactic definition and paste it over the call to the definition. If the tactic you are unfolding forms part of a larger tactic term, then you will need to bracket the unfolded term to ensure semantics preservation.

◇ Replace the tactic variables in the pasted term with the parameters you noted earlier.

◇ Replay your proofs to ensure they still work

Furthermore, if you wish to completely unfold a tactic and delete it, follow these additional steps:

- ◇ Ensure that the tactic you wish to unfold and delete is not recursive.
- ◇ Repeat the single step unfolding until there are no more instances.
- ◇ Then, apply the *delete unused tactic* refactoring to remove the old definition.

EXAMPLE Rather than provide an explicit example of this refactoring, I will look back at the fold tactic examples. Unfolding the tactic *newtac* in Listing 8.11 completely will transform the script into Listing 8.10, illustrating the symmetry of these two refactorings.

8.8 GENERALISE A TACTIC

DESCRIPTION Given a tactic definition like:

$$\textbf{tac}\ intros() \ := \ (impI \mid conjI \mid orI)\ ;\ ALL(intros) \mid ID$$

and imagine replacing the $(impI \mid conjI \mid orI)$ part with a tactic variable. The resulting tactic would be the more general *REPEAT* tactical of LCF:

$$\textbf{tac}\ intros(X) \ := \ X\ ;\ ALL(intros(X)) \mid ID$$

albeit with an incorrect name. Now, any proof that used *intros* would also have to be modified to instantiate its parameter. For example $\textbf{proof}(intros)$ becomes

$$\textbf{proof}(intros(impI \mid conjI \mid orI)).$$

Since this can be a bit harder to read, another option is to *fold* the instantiation as a new tactic definition:

$$\textbf{tac}\ intros \ := \ REPEAT(impI \mid conjI \mid orI)$$

which will also require a renaming of either the original definition or this new one.

The *generalise tactic* refactoring allows behavioural changes to be introduced in a structured way. The main condition on this refactoring is that the tactic expression being generalised does not contain any recursive calls. Since this expression is used as the value for the new parameter in any instances of the generalised tactic (to preserve semantics), any recursive call inside that expression must also be provided with a value for the new parameter. It is not clear what this new value should be.

MOTIVATION Generalisation is a good technique for extending the behaviour of currently defined objects. If you find you have multiple tactic terms that are similar but for one or two subterms, these are good candidates for generalisation.

RECIPE

- ◇ Create a copy of the tactic you would like to generalise, place it just above the old one and give it a fresh name.
- ◇ On this copied tactic, replace the term you would like to generalise over by a fresh tactic variable. Add it as a parameter to the tactic and also as a parameter to any recursive calls to the tactic.

- ◇ Check your proofs still replay.
- ◇ Replace the body of the original tactic with a direct call to the new tactic, with the parameter instantiated. Check the proofs still replay.
- ◇ Now, one-by-one, replace any calls to the old tactic with calls to the new one, checking that proofs replay after every substitution.
- ◇ Delete the old tactic and perform a renaming of the generalised tactic so that it is the same as before.

EXAMPLE The description above provides an example of the generalise tactic refactoring.

8.9 SUMMARY

This chapter further introduces proof refactorings by motivating and describing seven common refactorings. Each refactoring was also exemplified and a short 'recipe' was provided for performing the refactoring 'by hand'. In fact, all of the above refactorings have been formalised for the Hiscript language, and are described in the next two chapters.

9

REFACTORING PROOFS

9.1 INTRODUCTION

In this chapter and the next, I utilise the syntax for Hiscript that was introduced in Chapter 4 to give formal specifications for a variety of refactorings. Furthermore, I use the formal semantics to prove a correctness property: that these refactorings are *semantics preserving*. For the reader who has not read the first part, I provide a brief overview of the Hiscript framework in Appendix B.

CHAPTERS MAP In these chapters, I split the refactoring specifications into three parts:

1. In this chapter, in Sections 9.3 to 9.19, I specify refactorings that operate solely on declarative proofs: instances of the *prf* syntactic class for Hiscript.
2. Then, in Chapter 10, Section 10.2, I define refactorings that only affect one theory.
3. Finally, in Chapter 10, Section 10.3, I define refactorings that may make changes in multiple theories.

First though, in the next section, I formalise the notion of refactoring using the evaluation semantics for Hiscript. This chapter uses the syntax and semantics for Hiscript proofs that can be found in Figure 4.1 on page 46 (syntax) and Figure 4.3 on page 50 (semantics).

CONTRIBUTIONS These chapters contribute a collection of formally specified and provably correct refactorings for the Hiscript language; however, as well as this broad contribution, there are also a number of contributions that have arisen by dint of this work:

- Formal definitions for proof refactoring and semantics preservation are given for Hiscript.
- A collection of formally specified, correct refactorings are given for Hiscript.
- A notion of *range* for refactorings is introduced that allows refactorings to be specified in a more localised fashion.
- A small number of *patterns* for refactoring and techniques for proving them correct are identified.

9.2 A FORMAL DEFINITION OF REFACTORING

A 'traditional' refactoring, which I will write as $\xrightarrow{\mathcal{R}}$, acts on a *document*: a set of theories. If:

$$document \xrightarrow{\mathcal{R}} document'$$

then correctness (that is, *semantics preservation*) can be proved by relating the original and refactored proof document:

$$document \cong document'.$$

However, refactorings can have different ranges of transformation; that is, some make changes that are localised to a declarative proof: *flatten subproof* for example. Others modify a collection of theories, *rename lemma* for example. Using this observation, an appropriate *local* behaviour preservation property can be proved that can propagate up to become a correct *proof document refactoring*.

I now discuss each of these *levels* of refactoring.

TACTIC REFACTORING Let $\xrightarrow{tactic}$ be a transformation that is applied solely to an individual hitac expression, within a declarative proof, say. A transformation:

$$t \xrightarrow{tactic} t'$$

would preserve the external behaviour of a tactic expression, for a given list of input goals g and proof environment $(\mathcal{T}, \mathcal{L})$, if:

$$\langle g, t\rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle \quad \text{and} \quad \langle g, t'\rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s', g'\rangle$$

If a transformation has this property, it is *semantics preserving* and called a *tactic refactoring*. A tactic refactoring that behaves identically over all possible inputs is called *strongly semantics preserving*. Furthermore, a tactic refactoring can be applied to a subtactic and the correctness of the whole tactic expression can be argued by an induction on the structure of the evaluation rules. For example, this technique can be used for arguing that a statement:

```
apply t1 ; t2
```

can have t_2 refactored to t_2' if it behaves identically on the subgoals passed through to it by t_1.

PROOF REFACTORING

Next, a *proof refactoring* is a semantics preserving transformation of a declarative proof block **proof**...**qed**. I write it as $\xrightarrow{proof}$ and its correctness is defined with respect to a goal γ and a proof environment $(\mathcal{T}, \mathcal{L})$. If a transformation maps a proof $prf \xrightarrow{proof} prf'$ and prf evaluates as follows:

$$\langle \gamma, prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle$$

then it is semantics preserving if:

$$\langle \gamma, prf'\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s'\rangle.$$

That is, both evaluate successfully on the same goal. Furthermore if $s \vdash \gamma \longrightarrow g$ and $s' \vdash \gamma \longrightarrow g$ then I say it is *strongly semantics preserving*; thus, the resultant low-level proof thrusts out the same goals. All proof refactorings that I specify are strongly semantics preserving.

It is clear that applying a tactic refactoring inside a proof block is a correct proof refactoring.

SINGLE THEORY REFACTORING

Given a proof document, a single theory refactoring, written $\xrightarrow{theory}$, makes changes localised to one theory and will not affect any ancestor or descendant theories. The condition that ensures a transformation is a single theory refactoring is based on a property of the resulting proof environments. Let:

$$thy \xrightarrow{theory} thy'$$

and $\mathcal{D} \vdash thy \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$ and $\mathcal{D} \vdash thy' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$ for some theory map $\mathcal{D}$.

For the transformation behaviour to be independent of any other theory file, the only thing that can change is the hiproof in the lemma environment. That is:

$$\forall n.\ \mathcal{L}(n) = (visibility, \gamma, s) \Longrightarrow \mathcal{L}'(n) = (visibility, \gamma, s')$$

and, for tactic environments:

$$\forall n.\ \mathcal{T}(n) = (visibility, \overline{X}, t) \Longrightarrow \mathcal{T}'(n) = (visibility, \overline{X}, t)$$

While this may seem restrictive, it is important to note that it doesn't preclude $\mathcal{L}'$ and $\mathcal{T}'$ containing additional items. I will write this as $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{L} \subseteq_s \mathcal{L}'$, where $\subseteq_s$ means that the hiproof component of the map could be different. Apart from proof refactorings, an example of this type of refactoring is copying a lemma or tactic.

PROOF DOCUMENT REFACTORING

Finally, the most general refactoring may change all theories in a proof document. Recall from Section 5.3.2.1 that a proof document evaluation is a relation:

$$\vdash \mathcal{D} \Downarrow \mathcal{E}$$

which takes a theory map $\mathcal{D}$ (with a specified ordering) and is evaluated to construct a *proof environment map* $\mathcal{E}$. Let:

$$\mathcal{D} \xrightarrow{doc} \mathcal{D}'$$

be a transformation on theory maps. A *proof document refactoring* is a transformation satisfying one of the following three levels of semantics preservation that correspond to the possible modifications to a theory map:

EVALUATION PRESERVING. This is the weakest form for proof document refactorings and states that, if:

$$\vdash \mathcal{D} \Downarrow \mathcal{E}$$

then:

$$\vdash \mathcal{D}' \Downarrow \mathcal{E}'$$

for some $\mathcal{E}'$. That is, I only guarantee that the proof document still evaluates after the refactoring. An example of a refactoring that fits this category is *delete unused theory item* for a particular theory.

WEAK SEMANTICS PRESERVING. This form of correctness for refactoring states that at least the same lemmas are proved before and after the refactoring, although they may not be in the same theories as before and the resulting proofs may be different. An example of this category of refactoring is *move item between theories*. I represent this correctness property as follows: if:

$$\vdash \mathcal{D} \Downarrow \mathcal{E} \quad \text{and} \quad \vdash \mathcal{D}' \Downarrow \mathcal{E}'$$

then let:

$$(\mathcal{T}_{\mathcal{E}}, \mathcal{L}_{\mathcal{E}}) = \bigcup_{n \in \mathcal{E}} \mathcal{E}(n)$$

and:

$$(\mathcal{T}_{\mathcal{E}'}, \mathcal{L}_{\mathcal{E}'}) = \bigcup_{n \in \mathcal{E}'} \mathcal{E}'(n)$$

The refactoring is *weakly semantics preserving* if:

$$\forall n \in \mathcal{L}_{\mathcal{E}}.\ \mathcal{L}_{\mathcal{E}}(n) = (\gamma, s) \rightarrow \exists n' \in \mathcal{L}_{\mathcal{E}'}.\ \mathcal{L}_{\mathcal{E}'}(n') = (\gamma, s')$$

STRONG SEMANTICS PRESERVING. This correctness property extends the previous by restricting each theory to prove at least the same goals as previously. That is, if:

$$\vdash \mathcal{D} \Downarrow \mathcal{E} \quad \text{and} \quad \vdash \mathcal{D}' \Downarrow \mathcal{E}'$$

then for every theory n, if $\mathcal{E}(n) = (\mathcal{T}, \mathcal{L})$ and $\mathcal{E}'(n) = (\mathcal{T}', \mathcal{L}')$ then:

$$\forall l \in \mathcal{L}.\ \mathcal{L}(l) = (\gamma, s) \rightarrow \exists l' \in \mathcal{L}'.\ \mathcal{L}'(l') = (\gamma, s')$$

Refactorings that have this property are all single theory refactorings and the *rename item* refactoring.

9.3 COPY A HAVE STATEMENT

In Hiscript, the **have** statement introduces a local lemma, which exists only in the current proof block and any nested proof blocks. The **have** statements consist of a name, a goal, and a proof and are typically used for forward proof. One may wish to create a copy of a **have** statement, perhaps to experiment with a variant of the goal or the proof. This refactoring allows the proof developer to do so in a *controlled* way. Importantly, the copied statement in this refactoring is *not used*: no references to the original statement are changed to point to the duplicate.

This is one of the simplest refactorings that I demonstrate, but it is important for two reasons: it can be composed to create other refactorings; and, it serves as an easy refactoring in which to demonstrate the standard proof techniques and the structure of each refactoring specification.

Firstly, note that a proof refactoring is specified and proved correct with respect to a *goal* and a *proof environment*. In the following refactorings the goal will be written as γ and the proof environment as $(\mathcal{T}, \mathcal{L})$.

PARAMETERS All refactorings take parameters. The supplied parameters allow identification of the precise area to be refactored and also allow any necessary input from the user to be provided. In this example, the parameters are:

1. The proof block to refactor, *prf*.
2. The name of the **have** statement to copy, n.
3. The name for the copied **have** statement, m.

PRECONDITIONS The set of preconditions restrict the set of parameters for the refactoring to ensure that the crucial semantics preservation property holds. For copying a **have** statement, one needs to ensure:

1. The proof is *gap-free* and evaluates successfully:

   $$\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$$

   This precondition is true of all the refactorings in this chapter and will be omitted as a precondition for the rest.

2. The new name is *fresh*:

   $$m \notin (names(\mathcal{T}) \cup names(\mathcal{L}) \cup localnames(prf))$$

   That is to say, it is not already in the proof environment, nor defined locally in the supplied proof.

TRANSFORMATION RULES This refactoring is defined by a set of rules transforming the abstract syntax. The representation will be $x \xrightarrow{copyhave} y$. If the transformation is inductive (recursive), it is represented using an inference rule representation: where the transformations above the line must be performed to make the changes below the line. The rules are as follows:

$$\frac{stmts \xrightarrow{copyhave} stmts'}{\textsf{proof}(t)\ stmts\ \textsf{qed} \xrightarrow{copyhave} \textsf{proof}(t)\ stmts'\ \textsf{qed}} \quad \text{(CH-Proof)}$$

$$\frac{}{\begin{array}{l}\textsf{have}\ \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} \\ stmts\end{array} \xrightarrow{copyhave} \begin{array}{l}\textsf{have}\ \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} \\ \textsf{have}\ \mathfrak{m} :\ \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} \\ stmts\end{array}} \quad \text{(CH-Mod)}$$

$$\frac{(stmt \neq \textsf{have}\ \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf) \qquad stmts \xrightarrow{copyhave} stmts'}{\begin{array}{l}stmt \\ stmts\end{array} \xrightarrow{copyhave} \begin{array}{l}stmt \\ stmts'\end{array}} \quad \text{(CH-Stmt)}$$

The rules for this refactoring follow a pattern that many simple refactorings exhibit. I call it the *modifier rule* pattern. With this pattern, most of the rules simply recurse through the syntax until the part to be changed has been found (the rule named CH-Mod) — the modifier — then the rest of the proof is left unchanged.

CORRECTNESS To show that this refactoring specification preserves the semantics, the following theorem must be proved, for a given proof *prf*, have statement name $\mathfrak{n}$, and new name $\mathfrak{m}$, parameterised by γ and $(\mathcal{T}, \mathcal{L})$.

Theorem 24 (Correctness of Copy Have). *If* $\langle \gamma, prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$, *the preconditions hold, and* $prf \xrightarrow{copyhave} prf'$, *then* $\langle \gamma, prf' \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle$. *Furthermore, the refactoring is strongly semantics preserving:* $s' \vdash [\gamma] \longrightarrow []$.

Proof. This proof proceeds by analysing the structure of the transformation rules. Firstly, *prf* will be a proof block and will be transformed by CH-Proof. We know by the assumption that the left hand side of the transformation ($prf \equiv \textsf{proof}(\mathrm{t})\ stmts\ \textsf{qed}$) successfully evaluates and is evaluated by the rule B-Prf-Block:

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

We need to show that the right hand side ($prf' \equiv \textsf{proof}(\mathrm{t})\ stmts'\ \textsf{qed}$) also evaluates. Since we do not change the tactic *t*, we only need to show that *stmts'* evaluates successfully. To show this, we induct on the transformation rules on statements. The inductive case is the rule CH-Stmts, which follows simply by an appeal to the induction hypothesis and a case-analysis on the different type of statements (for *stmt*), but each case is trivial so we elide this detail. Finally then, we need to prove that if:

$$\langle \gamma, \textsf{have}\ \mathrm{n}{:}\ \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}}\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle \quad \text{and} \quad s \vdash [\gamma] \longrightarrow []$$

then:

$$\langle \gamma, \textsf{have}\ \mathrm{n}{:}\ \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}}\ \textsf{have}\ \mathrm{m}{:}\ \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}}\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle \quad \text{and} \quad s \vdash [\gamma] \longrightarrow []$$

The evaluation rule B-PRF-HAVE evaluates the left hand side of CH-MOD:

$$\frac{\langle [\gamma_n], prf_n\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad n \notin names(\mathcal{T}\cup\mathcal{L}) \qquad \langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[n\longmapsto(\gamma_n,\ s_1)])} \langle s\rangle}{\langle g, \textsf{have } n\text{: } \gamma_n\ prf_n\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle}$$

is used to evaluate these statements. On the right hand side of the transformation rule we get the result by two calls to B-PRF-HAVE and need to show that the following is a valid derivation:

$$\frac{\dfrac{\vdots}{\langle [\gamma_n], prf_n\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle} \quad n \notin names(\mathcal{T}\cup\mathcal{L}) \qquad \dfrac{\dfrac{\vdots}{\langle [\gamma_n], prf_n\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_1\rangle} \quad m \notin names(\mathcal{T}\cup\mathcal{L}') \qquad \dfrac{\vdots}{\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}'')} \langle s'\rangle}}{\langle g, \textsf{have } m\text{: } \gamma_n\ prf_n\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s'\rangle}}{\langle g, \textsf{have } n\text{: } \gamma_n\ prf_n\ \textsf{have } m\text{: } \gamma_n\ prf_n\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s'\rangle}$$

where $\mathcal{L}' = \mathcal{L}[n \longmapsto (\gamma_n,\ s_1)]$ and $\mathcal{L}'' = \mathcal{L}'[m \longmapsto (\gamma_n,\ s_1)]$. We know, by the assumptions that $\langle [\gamma_n], prf_n\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle$ and $n \notin names(\mathcal{T}) \cup names(\mathcal{L})$. We need to prove:

$$\langle g, \textsf{have } m\text{: } \gamma_n\ prf_n\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s'\rangle$$

which means we need to show:

- $\langle [\gamma_n], prf_n\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_1\rangle$. Now, $(\mathcal{T},\mathcal{L}')$ is a well-formed proof environment and $(\mathcal{T},\mathcal{L}) \subset (\mathcal{T},\mathcal{L}')$. We can then appeal to the environment extension theorem for proofs (Theorem 50 on page 219). We only need to show that:

  $$localnames(prf_n) \cap (names(\mathcal{T}) \cup names(\mathcal{L}')) = \varnothing.$$

  Since evaluation implies well-formedness (Theorem 12 on page 54), we have:

  $$localnames(prf_n) \cap (names(\mathcal{T}) \cup names(\mathcal{L})) = \varnothing.$$

  The only difference in environments is that $\mathcal{L}'$ contains the lemma n, but this cannot be in prf_n or it would not be well-formed. Thus, the intersection is still empty.

- $m \notin names(\mathcal{T}\cup\mathcal{L}')$. This is a direct consequence of precondition 2.

- $\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}'')} \langle s'\rangle$. By the assumption:

  $$\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s\rangle$$

  so an appeal to the closure of evaluation of a list of statements under environment extension (Theorem 51 on page 219) means we need to show:

  $$localnames(stmts) \cap (names(\mathcal{T}) \cup names(\mathcal{L}'')) = \varnothing.$$

Now, by precondition 2, we know that:

$$\mathfrak{m} \notin (\mathit{names}(\mathcal{T}) \cup \mathit{names}(\mathcal{L}) \cup \mathit{localnames}(\mathit{prf}))$$

Furthermore, $\mathit{localnames}(\mathit{stmts}) \subseteq \mathit{localnames}(\mathit{prf})$, so $\mathfrak{m} \notin \mathit{localnames}(\mathit{stmts})$.

Since $\mathit{localnames}(\mathit{stmts}) \cap (\mathit{names}(\mathcal{T}) \cup \mathit{names}(\mathcal{L}')) = \varnothing$ and the only difference between $\mathcal{L}'$ and $\mathcal{L}''$ is the addition of $\mathfrak{m}$, the result follows. The fact that $s' \vdash [\gamma] \longrightarrow []$ follows directly from the induction hypothesis.

□

In the forthcoming refactorings, the full proofs of correctness are provided in Appendix C and I elide the additional steps to show strong semantics preservation as they are usually straightforward.

9.4 DELETE UNUSED HAVE STATEMENT

If a **have** statement is not used where it is in scope, then it can be safely deleted. An element in the environment is not used if it is not a member of the minimal environment.

PARAMETERS The set of parameters for this refactoring are as follows:

1. The proof block being refactored, prf.
2. The name of the **have** statement to delete, $\mathfrak{n}$.

PRECONDITIONS The main precondition for this refactoring is that the statement is not used. That is:

$$\mathfrak{n} \notin \mathit{lemmas}(\mathit{prf})$$

where, recall from Section 4.5.2, the function *lemmas* returns the set of lemmas that are used in the proof (which also includes intermediate lemmas introduced by **have** statements).

TRANSFORMATION RULES As with the previous refactoring, this one follows the *modifier pattern*. Thus, only the modifier rule is shown:

$$\frac{}{\begin{array}{l}\textbf{have}\ \mathfrak{n} : \gamma_{\mathfrak{n}}\ \mathit{prf}_{\mathfrak{n}} \xrightarrow{\mathit{deletehave}} \\ \mathit{stmts} \qquad\qquad\qquad\qquad \mathit{stmts}\end{array}} \quad \text{(DH-Mod)}$$

CORRECTNESS The correctness of deleting a **have** statement with name $\mathfrak{n}$ from prf, parameterised by the goal γ and environment $(\mathcal{T}, \mathcal{L})$ is:

Theorem 25 (Correctness of Delete Have). *If $\langle \gamma, \mathit{prf} \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$, the preconditions hold, and $\mathit{prf} \xrightarrow{\mathit{deletehave}} \mathit{prf}'$, then $\langle \gamma, \mathit{prf}' \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$.*

Proof. See Appendix C, Section C.1. □

9.5 TACTIC SUBSTITUTION

Strictly speaking, this is not a proof refactoring, but a *tactic refactoring*. There are two variations:

- Substitute a tactic expression for a lemma application. For example, substituting a tactic t for the lemma *mylem* would be the following:

  $$\ldots lem\ mylem \ldots \xrightarrow{lemmasubst} \ldots t \ldots$$

- Substitute a tactic expression for a defined tactic. For example, substituting a tactic t for the tactic *mytac* would be the following:

  $$\ldots mytac(t_1, \ldots, t_n) \ldots \xrightarrow{tacsubst} \ldots t \ldots$$

  where $n \geqslant 0$ and the t_i are the parameters to the tactic.

This section describes the *substitute tactic* variation of this refactoring here. The *substitute lemma* version is identical.

PARAMETERS This refactoring takes three parameters:

- The tactic to refactor, t.
- The name of the defined tactic to remove, with a *fixed* set of parameters, $n(t_1, \ldots, t_n)$.
- The tactic to substitute, t_{sub}.

I will write the result of this refactoring using substitution syntax:

$$t[n(t_1, \ldots, t_n) := t_{sub}].$$

PRECONDITIONS Correctness of this refactoring follows from the equivalence of the tactic definition and the new tactic expression to replace it. I give here a strong precondition:

$$\forall g.\ \langle g, n(t_1, \ldots, t_n)\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g', s\rangle \ \rightarrow\ \langle g, t_{sub}\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g', s'\rangle$$

which says that for all possible goal lists: if the original defined tactic evaluates successfully, then so will the replacement expression, albeit with a possibly different resultant proof.

A much weaker precondition could be used in practice, which would say:

> at every instance of the tactic to be replaced, the replacement tactic must behave identically.

There is also an alternative to this refactoring where only the name of the defined tactic n is supplied. Then, correctness must be argued for all possible parameters or, in the weaker precondition case, for the parameters of each instance in the given hitac.

TRANSFORMATION RULES The transformation is a simple recursing through the structure of the tactic expression t with modification occurring when any instances of n are reached. I only show two of the rules:

$$\frac{t_1 \xrightarrow{tacsubst} t_1' \qquad t_2 \xrightarrow{tacsubst} t_2'}{t_1 \otimes t_2 \xrightarrow{tacsubst} t_1' \otimes t_2'} \quad \text{(TS-TENS)}$$

$$\frac{}{n(t_1, \ldots, t_n) \xrightarrow{tacsubst} t_{sub}} \quad \text{(TS-MOD)}$$

CORRECTNESS The correctness theorem is as follows:

Theorem 26 (Correctness of tactic substitution)**.** *If* $\langle g, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g', s\rangle$ *then*

$$\langle g, t[n(t_1, \ldots, t_n) := t_{sub}]\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g', s'\rangle.$$

Proof. The proof is a simple induction with appeals to the induction hypothesis in all cases except instances of the transformation rule TS-MOD, where the precondition is used. □

Apart from similar a correctness theorem, the *substitute lemma application* refactoring has an important property that I will use later:

Theorem 27 (Substitution ensures unused)**.** *For a tactic* t*, application of lemma named* l *and replacement tactic expression* t_{sub}*, if* $l \notin lemmas(t_{sub})$ *then* $l \notin lemmas(t[lem\ l := t_{sub}])$*. That is, substituting out a particular lemma means it no longer occurs in the tactic expression.*

Proof. The proof is a simple induction on the structure of the tactic. □

9.6 GENERAL TACTIC SUBSTITUTION

I can also define a more general notion of tactic substitution:

$$t[t_{sub} := t'_{sub}]$$

where a tactic expression t'_{sub} is substituted for any instances of the tactic expression t_{sub} in t. The rules and correctness requirements are identical: I require that the two tactic expressions behave identically under all possible lists of input goals.

9.7 SUBSTITUTION IN PROOFS

The notion of tactic and lemma substitution can be extended to operate on Hiscript proofs in a trivial way: by recursing through the proof block and applying the tactic substitution refactoring wherever tactic expressions are encountered.

As long as the tactic or lemma to substitute is not defined in the proof, it is straightforward to extrapolate the correctness argument to proofs and, furthermore, extend the substitution syntax as follows:

- *Tactic substitution in proofs:* is written $prf\,[n(t_1, \ldots, t_n) := t_{sub})]$.

- *Lemma substitution in proofs:* is written $prf\,[lem\ l := t_{sub}]$.

Again, substitution for lemmas has the following property:

Theorem 28 (Substitution ensures unused)**.** *For a proof prf, application of lemma named l and replacement tactic expression t_{sub}, if $\mathtt{l} \notin lemmas(t_{sub})$ then $\mathtt{l} \notin lemmas(prf\,[lem\ l := t_{sub}])$. That is, substituting out a particular lemma means it no longer occurs in the proof.*

Proof. The proof is again a simple induction on the structure of the proof, appealing to Theorem 27 where necessary. □

9.8 RENAME HAVE STATEMENT

The *rename have* refactoring is the first *composite* refactoring that I present. Instead of giving a set of transformation rules for this refactoring, I copy the original **have** statement and give it the new name. The new name can then be substituted for the old in any tactic expression that uses that lemma. Finally the original **have** statement can be deleted.

This modular approach allows for *composition* of refactorings: utilising the correctness proofs from the simple refactorings to demonstrate correctness of the composite.

PARAMETERS The parameters for this refactoring are the same as *copy have*:

- The proof to refactor, *prf*;
- The name of the lemma to rename, *old*;
- The new name for the lemma, *new*.

PRECONDITIONS The preconditions for this refactoring are again identical to *copy have*: the new name is *fresh*: $new \notin (names(\mathcal{T}) \cup names(\mathcal{L}) \cup localnames(prf))$.

TRANSFORMATION RULES Renaming a **have** statement with the name *old* to *new* in a proof *prf* could be written as:

$$\frac{prf \xrightarrow{copyhave} prf' \qquad prf'\,[lem\ old := lem\ new] \xrightarrow{deletehave} prf''}{prf \xrightarrow{renamehave} prf''}$$

CORRECTNESS Safe in the knowledge that the three refactorings being composed are correct, one must simply show that the preconditions are always satisfied:

1. The preconditions for *copy have* are satisfied directly by the preconditions of *rename have*; thus obtaining a refactored proof prf' which is well-formed and evaluates successfully to an environment $(\mathcal{T}, \mathcal{L}')$ where $\mathcal{L}'$ is the extension of $\mathcal{L}$ to include the lemma *new*.
2. To perform the substitution of the lemma application of *new* for an application of *old*, both lemmas must be shown to behave equivalently under all possible input goals, which is immediately true since the lemmas are identical.

3. Finally, I must show that the precondition for *delete have*, which will be:

   $$old \notin lemmas(prf'[lem\ old := lem\ new])$$

   is satisfied, which is a direct consequence of Theorem 28.

9.9 COPY, DELETE, RENAME TAC STATEMENT

Similar to the above refactorings, I can copy, delete, or rename a local tactic definition: the **tac** statement. The parameters and preconditions are identical. And, the transformation rules are similar; however, since tactics can be recursive, renaming must also be performed *inside* the tactic body:

$$\frac{t' = t[n := m]}{\begin{array}{l}\textbf{tac}\ n(\overline{X}) := t \\ stmts\end{array} \xrightarrow{copytac} \begin{array}{l}\textbf{tac}\ n(\overline{X}) := t \\ \textbf{tac}\ m(\overline{X}) := t' \\ stmts\end{array}} \quad \text{(CT-Mod)}$$

9.10 MERGING PROCEDURAL STEPS

This is a small set of refactorings that can utilise tactic equalities to merge **apply** statements (or procedural proof steps). There are two variations that can be performed:

- *Merge procedural steps.* Transforming a proof like:

  proof(t)
  …
  apply t_1
  apply t_2
  …
  qed

  into

  proof(t)
  …
  apply t_1 ; t_2
  …
  qed

- *Merge procedural step with introduction tactic.* Behaving similarly and transforming

  proof(t)
  apply t_1
  …
  qed

  into

proof(t ; t_1)
  ...
qed

The transformation is simple and correctness easy to show.

PARAMETERS *Merge procedural steps* takes two parameters:

- The proof block to refactor, *prf*.
- The *second* apply step: **apply** t_2 to match with.

PRECONDITIONS There are no preconditions.

TRANSFORMATION RULES The first refactoring *merge procedural steps* is an instance of the modifier pattern, with the modifier rule being:

$$\frac{}{\begin{array}{l}\textbf{apply}\ t_1\\ \textbf{apply}\ t_2\\ stmts\end{array} \xrightarrow{mergeprocstep} \begin{array}{l}\textbf{apply}\ t_1\ ;\ t_2\\ stmts\end{array}} \quad \text{(MPS-Mod)}$$

Merging at the top, is slightly different:

$$\frac{}{\begin{array}{l}\textbf{proof}(t)\\ \quad \textbf{apply}\ t_1\\ \quad stmts\\ \textbf{qed}\end{array} \xrightarrow{mergeproctop} \begin{array}{l}\textbf{proof}(t\ ;\ t_1)\\ \quad stmts\\ \textbf{qed}\end{array}} \quad \text{(MPT-Mod)}$$

CORRECTNESS It is simple to see that the evaluation semantics for proof blocks and for procedural steps both utilise the sequencing operator to compose proofs.

9.11 SWAPPING STATEMENTS

Consider a proof block:

proof(t)
  ...
  $stmt_1$
  $stmt_2$
  ...
qed

One can swap the order of certain types of statements. For example, two **show** statements can never be swapped as they need to be provided in order of the remaining goals, but two **have** statements can be swapped provided one does not need the other in its proof. An **apply** step can always be moved above any **show** statement in a way that is described below.

I will only show a few of these transformation rules. This refactoring is defined without preconditions and parameterised by:

- The proof block, *prf*.
- The bottom statement $stmt_{up}$; the one to be moved upwards.

TRANSFORMATION RULES This refactoring also follows the modifier pattern, except this time there are a set of modifying transformation rules: one for each possible pair of statements. There is then a *default* rule for the case that no others are applicable: two apply statements, for instance. I give a few of these rules below:

$$\frac{stmt_{up} \equiv \textsf{have } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \qquad \mathfrak{n} \notin lemmas(prf_{\mathfrak{m}})}{\begin{array}{ll}\textsf{have } \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} & \textsf{have } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \\ \textsf{have } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \xrightarrow{swapstmt} & \textsf{have } \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} \\ stmts & stmts\end{array}} \text{(SS-MOD-HAVEHAVE)}$$

$$\frac{stmt_{up} \equiv \textsf{apply } t}{\begin{array}{ll}\textsf{show } \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} & \textsf{apply } id \otimes t \\ \textsf{apply } t \xrightarrow{swapstmt} & \textsf{show } \mathfrak{n} : \gamma_{\mathfrak{n}}\ prf_{\mathfrak{n}} \\ stmts & stmts\end{array}} \text{(SS-MOD-SHOWAPP)}$$

$$\frac{stmt_{up} \equiv \textsf{show } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \qquad \mathfrak{n} \notin tacs(\mathfrak{m})}{\begin{array}{ll}\textsf{tac } \mathfrak{n}(\overline{X}) := t & \textsf{show } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \\ \textsf{show } \mathfrak{m} : \gamma_{\mathfrak{m}}\ prf_{\mathfrak{m}} \xrightarrow{swapstmt} & \textsf{tac } \mathfrak{n}(\overline{X}) := t \\ stmts & stmts\end{array}} \text{(SS-MOD-TACSHOW)}$$

$$\frac{}{\begin{array}{ll}stmt_1 & stmt_1 \\ stmts_2 \xrightarrow{swapstmt} & stmt_2 \\ stmts & stmts\end{array}} \text{(SS-MOD-DEFAULT)}$$

CORRECTNESS

Theorem 29 (Correctness of Swap Statements). *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$, *the preconditions hold, and* $prf \xrightarrow{swapstmt} prf'$, *then* $\langle \gamma, prf' \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$.

Proof. The proof is detailed in Section C.2 of Appendix C. □

9.12 BACKWARDS STYLE PROOF TO FORWARDS STYLE PROOF

I now move on to refactorings that make more ambitious changes to a proof block. The *backwards to forwards* refactoring transforms a proof that is in the form of Listing 9.1 to the form of Listing 9.2.

```
proof(t)
  show goal_1: γ1
    prf_1
   ...
  apply t_1
   ...
  show goal_n : γn
    prf_n
qed
```

Listing 9.3: Proof block with apply statement

```
proof(t)
  show goal_1: γ1
    prf_1
   ...
  show goal_n : γn
    prf_n
qed
```

Listing 9.1: Initial proof block

```
proof
  have goal_1: γ1
    prf_1
   ...
  have goal_n : γn
    prf_n
  from goal_1 ... goal_n show γ by t
qed
```

Listing 9.2: Forward proof block

PARAMETERS In the basic form, this refactoring takes a single parameter *prf*: the proof block to refactor. Since show statements can also occur without a name, this refactoring could be generalised to take a list of names as a second parameter, but I do not cover this case here.

PRECONDITIONS This refactoring has two preconditions:

1. The names of each show statement in the proof block are fresh.
2. There are no apply statements within the proof block. To see why, consider the proof block in Listing 9.3. The procedural step in the middle acts on *all* of the remaining goals; then, if I transform all the show statements to have statements, the uses of the lemmas after that procedural step will not match the goals introduced by the tactic t. This is not a fundamental limitation; it is possible to apply the *swap statements* and *merge procedural steps* refactorings to move any apply steps above all of the show statements.

TRANSFORMATION RULES The following rules perform this refactoring. Each statement is transformed independently:

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{proof}(t) & \textsf{proof} \\ stmts & \xrightarrow{back2forward} \quad stmts_{b2f} \\ \textsf{qed} & \textsf{qed}\end{array}} \quad \text{(B2F-Prf)}$$

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{show } n_\gamma : \gamma\ prf & \xrightarrow{back2forward} \quad \textsf{have } n_\gamma : \gamma\ prf \\ stmts & stmts_{b2f}\end{array}} \quad \text{(B2F-Show)}$$

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{from } n_1 \ldots n_n \textsf{ show } n_\gamma : \gamma \textsf{ by } t & \xrightarrow{back2forward} \quad \textsf{from } n_1 \ldots n_n \textsf{ have } n_\gamma : \gamma \textsf{ by } t \\ stmts & stmts_{b2f}\end{array}} \quad \text{(B2F-FromShow)}$$

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{have } n_\gamma : \gamma\ prf & \xrightarrow{back2forward} \quad \textsf{have } n_\gamma : \gamma\ prf \\ stmts & stmts_{b2f}\end{array}} \quad \text{(B2F-Have)}$$

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{from } n_1 \ldots n_n \textsf{ have } n_\gamma : \gamma \textsf{ by } t & \xrightarrow{back2forward} \quad \textsf{from } n_1 \ldots n_n \textsf{ have } n_\gamma : \gamma \textsf{ by } t \\ stmts & stmts_{b2f}\end{array}} \quad \text{(B2F-FromHave)}$$

$$\frac{stmts \xrightarrow{back2forward} stmts_{b2f}}{\begin{array}{ll}\textsf{tac } n(\overline{X}) := t & \xrightarrow{back2forward} \quad \textsf{tac } n(\overline{X}) := t \\ stmts & stmts_{b2f}\end{array}} \quad \text{(B2F-Tac)}$$

$$\frac{[n_1, \ldots, n_k] = shows(prf) \qquad t = introtac(prf)}{[] \xrightarrow{back2forward} \textsf{from } n_1 \ \ldots \ n_k \textsf{ show } \gamma \textsf{ by } t} \quad \text{(B2F-Empty)}$$

For the final transformation rule, I make an assumption that the goal that the refactoring is parameterised by is γ. I also use $[]$ to represent the empty list of statements.

CORRECTNESS

Theorem 30 (Correctness of backward to forward refactoring). *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$, *the preconditions hold, and* $prf \xrightarrow{back2forward} prf'$, *then* $\langle \gamma, prf' \rangle \Downarrow_{\mathcal{E}} \langle s' \rangle$.

Proof. Informally, note that the main goal is preserved throughout the list of refactored statements, since only a show actually modifies the list of goals. Thus, when the final from statement is reached, it is operating on the correct goal. And, by the semantics of from, each of the n_i must solve the ith subgoal generated by the tactic t. This is exactly what each of the show statements did; thus, the behaviour is the same as these now exist as lemmas in the proof. A formal proof of this theorem is given in Section C.3 of Appendix C. □

9.13 FORWARDS STYLE PROOF TO BACKWARDS STYLE PROOF

Refactoring can go the other way. This refactoring takes a statement:

from $n_1 \ldots n_n$ show γ by t

and transforms it to a statement:

show γ
proof(t)
  show γ_1 by *lem* n_1
  ...
  show γ_n by *lem* n_n
qed

Although it may seem that the proofs of each lemma are not very interesting, I can use a refactoring called *unfold proof* that is defined below to provide the original declarative proof of that lemma. Then, if any of the lemmas n_i are now unused, they can be deleted using the refactoring *delete unused have* or the more global refactoring *delete unused lemma*. Furthermore, correctness of this refactoring is simple to justify and its correctness proof is an instance of the *modifier* pattern.

9.14 DECLARATIVE TO PROCEDURAL

Declarative proofs increase the robustness and understandability of a proof; however, they can also be very long and, perhaps, tedious to read, particularly if the proof is 'trivial' for a human. In these cases, procedural proofs are shorter and can do the job just as well. One advantage of the formal semantics for Hitac and Hiscript is that I can relate one to the other, thus allowing a declarative proof to be 'collapsed' into a hitac that does the same job.

There are two versions of this refactoring: the simplest would transform the following proof:

proof(t)
  show γ_1 prf_1
  ...
  show γ_n prf_n
qed

into

by t ; $t_1 \otimes \ldots \otimes t_n$

where each t_i is the recursive invocation of the transformation on prf_i. Here the restriction is that the proof is completely backward: there are no **have** (or **tac**) statements.

If not, the proof does not have such a straightforward transformation. Consider a proof:

proof(t)
  show γ_1 prf_1
  ...
  have *lem*: *goal* *prf*
  ...
  show γ_n prf_n
qed

and here the best that can be done is refactor it to

proof
  have *lem*: *goal* t_{lem}
  apply t ; $t_1 \otimes \ldots \otimes t_n$
qed

but the situation is complicated when there are nested **have** or **tac** statements in any prf_i. I describe the basic transformation here and plan to investigate a more sophisticated transformation as future work.

PARAMETERS This refactoring takes a proof block, *prf*.

PRECONDITIONS As well as the precondition that there are no environment modifying statements like **have** or **tac**, this refactoring will succeed as long as there are no **gap** statements.

TRANSFORMATION RULES The set of transformation rules is given to map a *prf* into a hitac t. The top-level rule is:

$$\frac{prf \xrightarrow{dec2proc} t_{prf}}{prf \xrightarrow{dec2procfull} \textbf{by}\ t_{prf}} \quad \text{(DecToProc)}$$

which uses the following rules to recursively fold nested subproofs:

$$\frac{stmts \xrightarrow{dec2proc} t_{stmts}}{\begin{array}{l}\textbf{proof}(t) \xrightarrow{dec2proc} t\ ;\ t_{stmts} \\ \quad stmts \\ \textbf{qed}\end{array}} \quad \text{(D2P-Proof)}$$

$$\frac{prf \xrightarrow{dec2proc} t_{prf} \qquad stmts \xrightarrow{dec2proc} t_{stmts}}{\textsf{show}\ \mathfrak{n} : \gamma\ prf \xrightarrow{dec2proc} t_{prf} \otimes t_{stmts}\ stmts} \tag{D2P-Show}$$

$$\frac{stmts \xrightarrow{dec2proc} t_{stmts}}{\textsf{from}\ \mathfrak{n}_1 \ldots \mathfrak{n}_m\ \textsf{show}\ \mathfrak{n} : \gamma\ \textsf{by}\ t \xrightarrow{dec2proc} (t\ ;\ (lem\ n_1 \otimes \ldots \otimes lem\ n_m)) \otimes t_{stmts}\ stmts} \tag{D2P-FromShow}$$

$$\frac{stmts \xrightarrow{dec2proc} t_{stmts}}{\textsf{apply}\ t \xrightarrow{dec2proc} t\ ;\ t_{stmts}\ stmts} \tag{D2P-Apply}$$

$$\frac{}{[] \xrightarrow{dec2proc} \langle\rangle} \tag{D2P-Empty}$$

CORRECTNESS To show correctness of this refactoring, I (essentially) show that the tactic returned as a result of the transformation rules solves the same goals as the original proof. That is:

Theorem 31 (Correctness of declarative to procedural). *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$ *and* $prf \xrightarrow{dec2proc} t_{prf}$, *then* $\langle \gamma, t_{prf} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s' \rangle$.

Proof. The proof is given in Section C.4 of Appendix C. □

9.15 PROCEDURAL TO DECLARATIVE

Conversely, procedural proofs can be expanded, creating a declarative representation of them. The approach here is, rather than translate the hitac in by t, to translate the hiproof.

9.15.1 *Hiproofs to Hiscript*

Recall the definition of Tensor Normal Form (TNF) for a hiproof from Section 2 on page 23. This normal form captures hiproofs of the form $s\ ;\ s_1 \otimes \ldots \otimes s_n$, which is exactly the form that the evaluation semantics for proof blocks takes:

```
proof(t)
  show γ1 prf1
  ...
  show γn prfn
qed
```

The following transformation rules cover each case in the definition of TNF. Note that there is no rule for the empty hiproof $\langle\rangle$ as it does not correspond to a Hiscript proof. This refactoring makes sense only with respect to a list of goals (that the hiproof validates and the resultant Hiscript proof solves), which is represented 'above the line' using the hiproof validation relation.

$$\frac{id \vdash \gamma \longrightarrow \gamma}{id \xrightarrow{proc2dec} \textsf{show}\ \gamma \quad \textsf{gap}} \tag{P2D-Id}$$

$$\frac{a \vdash \gamma \longrightarrow []}{a \xrightarrow{proc2dec} \textsf{show}\ \gamma\ \textsf{by}\ a} \tag{P2D-Atomic1}$$

$$\frac{a \vdash \gamma \longrightarrow [\gamma_1, \ldots, \gamma_n]}{a \xrightarrow{proc2dec} \begin{array}{l}\textsf{show}\ \gamma \\ \textsf{proof}(a) \\ \textsf{show}\ \gamma_1 \\ \quad \textsf{gap} \\ \ldots \\ \textsf{show}\ \gamma_n \\ \quad \textsf{gap} \\ \textsf{qed}\end{array}} \tag{P2D-Atomic2}$$

$$\frac{[l]\ s \vdash \gamma \longrightarrow g \qquad s \xrightarrow{proc2dec} prf}{[l]\ s \xrightarrow{proc2dec} \begin{array}{l}\textsf{show}\ \gamma \\ [l]\ prf\end{array}} \tag{P2D-Lab}$$

$$\frac{a\ ;\ s \vdash \gamma \longrightarrow g \qquad s \xrightarrow{proc2dec} stmts}{a\ ;\ s \xrightarrow{proc2dec} \begin{array}{l}\textsf{show}\ \gamma \\ \textsf{proof}(a) \\ \quad stmts \\ \textsf{qed}\end{array}} \tag{P2D-Seq}$$

$$\frac{s_1 \otimes \ldots \otimes s_n \vdash [\gamma_1, \ldots, \gamma_n] \longrightarrow g \qquad s_1 \xrightarrow{proc2dec} prf_1 \quad \ldots \quad s_n \xrightarrow{proc2dec} prf_n}{s_1 \otimes \ldots \otimes s_n \xrightarrow{proc2dec} \begin{array}{l} \textsf{show } \gamma_1 \\ \quad prf_1 \\ \ldots \\ \textsf{show } \gamma_n \\ \quad prf_n \end{array}} \text{(P2D-Tens)}$$

$$\frac{([l]\ s_1)\ ;\ s_2 \vdash \gamma \longrightarrow g \qquad s_2 \xrightarrow{proc2dec} stmts}{([l]\ s_1)\ ;\ s_2 \xrightarrow{proc2dec} \begin{array}{l} \textsf{show } \gamma \\ \textsf{proof}([l]\ s_1) \\ \quad stmts \\ \textsf{qed} \end{array}} \text{(P2D-LabSeq)}$$

I have to cheat a little in the last transformation and consider $[l]\ s_1$ as one tactic. The reason behind the cheat is that Hiscript does not explicitly offer a construct for introducing a labelled proof block that leaves holes to be filled later.

CORRECTNESS The correctness argument for this refactoring uses the correspondence between hitacs and hiproofs to consider the hiproof s a hitac in the sense that if $s \vdash g_1 \longrightarrow g_2$ then

$$\langle g_1, s\rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle g_2, s\rangle.$$

where, $(\{\},\{\})$ is the empty environment; no defined tactics or lemmas will be used.

Theorem 32 (Correctness of Hiproof to Hiscript)**.** *For a hiproof s in TNF, if $s \xrightarrow{proc2dec} prf$*

$$\langle g, s\rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle g', s\rangle$$

then

$$\langle g, prf\rangle \Downarrow_{(\{\},\{\})} \langle s'\rangle$$

Proof. The proof of this theorem is given in Section C.5 of Appendix C. □

9.15.2 *More sensitive transformations*

I can also utilise the hierarchy to provide contextual information in order to construct a 'higher-level' proof. To see how, consider the TNF case:

$$s \equiv ([l]\ s')\ ;\ s'''$$

if I have a proof environment $(\mathcal{T}, \mathcal{L})$ with any items with the name used in that label, I can replace the whole labelled hiproof with a reference to that defined tactic or lemma. The two transformation rules that accomplish this extension are as follows:

$$\frac{([lem\ l]\ s_1)\ ;\ s_2 \vdash \gamma \longrightarrow g \qquad \mathcal{L}(l) = (\gamma, s_1) \qquad s_2 \xrightarrow{proc2dec} stmts}{([lem\ l]\ s_1)\ ;\ s_2 \xrightarrow{proc2dec} \begin{array}{l}\textbf{show}\ \gamma \\ \textbf{proof}(lem\ l) \\ \quad stmts \\ \textbf{qed}\end{array}} \text{(P2D-LabLem)}$$

$$\frac{([mytac]\ s_1)\ ;\ s_2 \vdash \gamma \longrightarrow g \qquad \mathcal{T}(mytac) = ([], t) \qquad s_2 \xrightarrow{proc2dec} stmts}{([mytac]\ s_1)\ ;\ s_2 \xrightarrow{proc2dec} \begin{array}{l}\textbf{show}\ \gamma \\ \textbf{proof}(mytac) \\ \quad stmts \\ \textbf{qed}\end{array}} \text{(P2D-LabTac)}$$

Note that I currently do not allow tactics with parameters to be included. The reason behind this is that there is currently no obvious way to detect what the appropriate instantiation for those parameters may be.

9.16 FLATTEN SUBPROOF

Consider a purely forward proof block (the proof of $P \wedge Q \vdash Q \wedge P$):

```
show P ∧ Q ⇒ Q ∧ P
proof (impI)
  show P ∧ Q ⊢ Q ∧ P
  proof
    have q: P ∧ Q ⊢ Q by conjE ; ax
    have p: P ∧ Q ⊢ P by conjE ; ax
    from q p show P ∧ Q ⊢ Q ∧ P by conjI
  qed
qed
```

within the context of a parent proof block This evaluates identically with the following:

```
show P ∧ Q ⇒ Q ∧ P
proof (impI)
  have q: P ∧ Q ⊢ Q by conjE ; ax
  have p: P ∧ Q ⊢ P by conjE ; ax
  from q p show P ∧ Q ⊢ Q ∧ P by conjI
qed
```

which is more natural. This type of refactoring, removing a proof block, is called *flatten subproof* and can be applied to more general proof blocks. This is also the first proof refactoring that I have seen that requires two proof blocks: an inner and outer. This refactoring operates more generally on inner proof blocks that do not contain any apply statements, mapping the left proof block to the right proof block:

```
proof                          proof
...                            ...

  show n: γ                      apply t'
  proof(t)                       stmt₁
    stmt₁                        ...
    ...                          stmtₙ
    stmtₙ
  qed

...                            ...
qed                            qed
```

where t' is a modification of the proof block introduction tactic t.

PARAMETERS This refactoring takes two parameters:

- The outer proof block, prf_{out}.
- The statement proved by the inner proof block to flatten, $stmt_{in}$. This will be a **show** statement. I write the proof block as prf_{in}.

PRECONDITIONS There are two preconditions to this refactoring:

1. As already mentioned, there must be no **apply** statements in prf_{in}.
2. Furthermore, none of the locally defined lemmas or tactics in $stmt_{in}$ must clash with any in the rest of the outer proof block.

The final precondition is required because any of the local names introduced in prf_{in} would previously not be in scope for all the statements after the statement that was proved with prf_{in}, but after it is flattened, they will be in scope.

TRANSFORMATION RULES I only give the rule for transforming the inner proof block. The other rules are simply instances of the modifier pattern.

$$\frac{prf \xrightarrow{flattensubproof} stmts}{stmt_{in} \equiv \textbf{show}\ \gamma \quad prf_{in} \xrightarrow{flattensubproof} stmts} \quad \text{(FS-Mod)}$$

$$\frac{}{\begin{array}{ll}\textbf{proof}(t) & \textbf{apply}\ t \otimes ID \\ \quad stmts & \xrightarrow{flattensubproof} \quad stmts \\ \textbf{qed} & \end{array}} \quad \text{(FS-Prf1)}$$

where ID is a hitac tactic that applies the identity proof to all goals. For the case where $t \equiv id$, I simply omit the **apply** step:

$$\frac{}{\begin{array}{l}\textsf{proof} \\ \quad \mathit{stmts} \\ \textsf{qed}\end{array} \xrightarrow{\mathit{flattensubproof}} \mathit{stmts}} \quad \text{(FS-Prf2)}$$

CORRECTNESS

Theorem 33 (Correctness of flatten subproof). *If* $\langle \gamma, \mathit{prf}_{out} \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ *and the transformation maps* $\mathit{prf}_{out} \xrightarrow{\mathit{flattensubproof}} \mathit{prf}'_{out}$ *then* $\langle \gamma, \mathit{prf}_{out'} \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle$.

Proof. A proof of this theorem is given in Section C.6 of Appendix C. □

9.17 FOLDING AND UNFOLDING

In this section, I detail four similar refactorings:

- *Fold a tactic expression*. This refactoring takes a tactic expression as a parameter and a fresh name and creates a new local tactic definition with that expression as the body of the definition. The supplied reference to that tactic expression is then replaced by an instance of the defined tactic.
- *Fold a proof*. Similarly, this refactoring takes a Hiscript proof and transforms it to a local lemma i.e. a **have** statement.
- *Unfold a tactic*. Takes the name of a defined tactic as a parameter and replaces any calls to this definition with the body, substituting formal parameters for the appropriate instances.
- *Unfold a proof*. Similarly, this refactoring replaces any instances of a lemma application with its proof body.

9.17.1 *Fold a tactic*

Inside a proof block, a tactic expression can occur in three places:

1. As part of the proof block introduction tactic.
2. As part of a procedural step.
3. As part of a **from** statement.

For each case, the folded tactic definition will be introduced above each of these constructs and the *general tactic substitution* refactoring will be performed to replace the original expression with the new definition.

PARAMETERS This refactoring takes three parameters:

- The tactic expression to fold, t_{fold}.
- The name for the newly created tactic definition, *tname*.
- The proof block to refactor, *prf*.
- The statement inside *prf* containing the expression t_{fold}, stmt_{fold}.

PRECONDITIONS The precondition for this refactoring is that the new name *tname* is fresh in the proof block being refactored.

TRANSFORMATION RULES This refactoring follows the modifier pattern and has a case for each of the three possible values for the statement containing the expression $stmt_{fold}$. I only show the rule for refactoring a procedural step:

$$\frac{}{\begin{array}{l} stmt_{fold} \equiv \textsf{apply } t \\ stmts \end{array} \xrightarrow{foldtac} \begin{array}{l} \textsf{tac } tname := t_{fold} \\ \textsf{apply } t[t_{fold} := tname] \\ stmts \end{array}} \text{(FT-APPLY)}$$

CORRECTNESS The correctness theorem for this refactoring follows the modifier pattern and has a straightforward proof that I do not detail here.

9.17.2 *Unfold a tactic*

PARAMETERS AND PRECONDITIONS I *unfold a defined tactic* called *tname* in a proof block *prf*. There are no preconditions for this refactoring.

TRANSFORMATION RULES This refactoring is very similar to simple tactic substitution in proofs; however, there is one slight complication: it is possible that the tactic *tname* is defined in *prf*:

```
proof
  ...
  tac tname(X) := ...
  ...
qed
```

inside which the unfolding cannot occur. Furthermore, if *tname* has any parameters, I must perform the appropriate substitution during the unfolding.

The transformation is performed with respect to some particular environment $(\mathcal{T}, \mathcal{L})$ where, in particular, I know that:

$$\mathcal{T}(tname) = (t_{tname}, \overline{X})$$

A selection of the transformation rules is shown:

$$\frac{stmts \xrightarrow{unfoldtac} stmts' \qquad t' = t[tname(\overline{t}) := t_{tname}[\overline{X} := \overline{t}]]}{\begin{array}{l} \textsf{proof}(t) \\ stmts \\ \textsf{qed} \end{array} \xrightarrow{unfoldtac} \begin{array}{l} \textsf{proof}(t') \\ stmts' \\ \textsf{qed} \end{array}} \text{(UT-PRF)}$$

$$\frac{stmts \xrightarrow{unfoldtac} stmts' \qquad t' = t[tname(\overline{t}) := t_{tname}[\overline{X} := \overline{t}]]}{\begin{array}{l} \textsf{apply } t \\ stmts \end{array} \xrightarrow{unfoldtac} \begin{array}{l} \textsf{apply } t' \\ stmts' \end{array}} \text{(UT-APPLY)}$$

$$\frac{stmts \xrightarrow{unfoldtac} stmts'}{\begin{array}{l}\textbf{tac}\ tname(\overline{X})\ :=\ t_{tname} \\ stmts\end{array} \xrightarrow{unfoldtac} \begin{array}{l}\textbf{tac}\ tname(\overline{X})\ :=\ t_{tname} \\ stmts'\end{array}} \quad \text{(UT-TAC1)}$$

$$\frac{stmts \xrightarrow{unfoldtac} stmts' \qquad t' = t[tname(\overline{t}) := t_{tname}[\overline{X} := \overline{t}]]}{\begin{array}{l}\textbf{tac}\ n(\overline{X})\ :=\ t \\ stmts\end{array} \xrightarrow{unfoldtac} \begin{array}{l}\textbf{tac}\ n(\overline{X})\ :=\ t' \\ stmts'\end{array}} \quad \text{(UT-TAC2)}$$

CORRECTNESS This refactoring can be shown correct by an induction on the transformation and appealing to the tactic substitution refactoring correctness property where necessary.

It is also interesting to note that if:

1. the tactic is not recursive;
2. and, the tactic is defined in *prf*;

then after this refactoring it will no longer be used. That is, one could then perform a *delete unused tactic* refactoring.

9.17.3 *Fold a proof*

Folding a proof is analagous to folding a tactic. This refactoring extracts a proof as a local lemma, which is the used directly.

PARAMETERS AND PRECONDITIONS The refactoring takes three parameters:

- The proof *prf* to refactor.
- The **show** statement whose proof is folded: **show** γ_{fold} prf_{fold}.
- The name for the newly created local lemma, *l*.

The only precondition for this refactoring is that the new name *lem* is fresh:

$$l \notin localnames(prf)$$

TRANSFORMATION RULES This transformation follows the modifier pattern. The important rule is:

$$\frac{}{\begin{array}{l}\textbf{show}\ \gamma_{fold} \\ \quad prf_{fold} \\ stmts\end{array} \xrightarrow{foldproof} \begin{array}{l}\textbf{have}\ l :\ \gamma_{fold} \\ \quad prf_{fold} \\ \textbf{show}\ \gamma_{fold}\ \textbf{by}\ lem\ l \\ stmts\end{array}} \quad \text{(FP-MOD)}$$

CORRECTNESS The correctness of this refactoring is straightforward to show by an induction. The main work of the proof is to show that, given that the derivation below is valid:

$$\frac{\dfrac{\vdots}{\langle \gamma_{fold}, prf_{fold}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle} \qquad \dfrac{\vdots}{\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}}{\langle \gamma_{fold} :: g,\ \textsf{show}\ \gamma_{fold}\ prf_{fold}\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle ([show]\ s_1) \otimes s_2\rangle}$$

then the following refactored statement list has a valid derivation:

$$\frac{\dfrac{\vdots}{\langle \gamma_{fold}, prf_{fold}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle} \qquad \dfrac{\dfrac{\vdots}{\langle \gamma_{fold}, \textsf{by}\ lem\ l\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_2\rangle} \qquad \dfrac{\vdots}{\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_3\rangle}}{\langle \gamma_{fold} :: g,\ \textsf{show}\ \gamma_{fold}\ \textsf{by}\ lem\ l\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle ([show]\ s_2) \otimes s_3\rangle}}{\langle \gamma_{fold} :: g,\ \textsf{have}\ l :\ \gamma_{fold}\ prf_{fold}\ \ \textsf{show}\ \gamma_{fold}\ \textsf{by}\ lem\ l\ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle ([show]\ s_2) \otimes s_3\rangle}$$

where $\mathcal{L}' = \mathcal{L}[l \mapsto (\gamma_{fold}, s_1)]$, which is straightforward.

9.17.4 *Unfold a proof*

Unfolding a proof has a more complicated behaviour as lemma applications can occur deep in a tactic expression. It is easy to define and prove the inverse operation to *fold a proof*. That is, a transformation of by *lem l* into a proof block replicated from the definition of the lemma l and I do not give details of it here; however, since this cannot happen in general, another approach is necessary. The choice I take is to use the *declarative to procedural* refactoring to turn the proof of the lemma l into a tactic expression, then substitute.

PARAMETERS AND PRECONDITIONS Again, I have three parameters:

- The proof block to perform the unfolding in prf.
- The lemma name that I want to unfold: l.
- The original proof of the lemma: prf_l. This is currently not stored in the lemma environment so it must be provided as a parameter.

TRANSFORMATION RULES AND CORRECTNESS I can simply perform this refactoring by a lemma substitution in the proof:

$$\frac{prf_l \xrightarrow{dec2proc} t_l}{prf \xrightarrow{unfoldproof} prf\,[lem\ l := t_l]}$$

Correctness of this refactoring, then comes directly from correctness of its two component refactorings.

9.18 GENERALISE A TACTIC

The penultimate refactoring in this chapter is a composite refactoring. To generalise a tactic means to transform a proof block that looks like:

```
proof
  tac tname := t1 ; t2 ; t3
  ...
qed
```

into a proof block that looks like:

```
proof
  tac tnamegen(X) := t1 ; X ; t3
  tac tname := tnamegen(t2)
  ...
qed
```

PARAMETERS This refactoring requires the following parameters:

- The proof block to refactor, prf.
- The tactic expression t_{gen} that I abstract over.
- The name of the tactic that I wish to generalise, n.
- The name of the new, more general tactic, n_{gen}.
- Finally, the name of the tactic variable that will be introduced, X_{gen}.

PRECONDITIONS This refactoring requires that:

1. The name for the general tactic is fresh:

   $$n_{gen} \notin localnames(prf)$$

2. The new tactic variable is also fresh:

   $$X_{gen} \notin vars(n)$$

3. The tactic expression to generalise over does not contain any variables:

   $$vars(t_{gen}) = \varnothing$$

4. The tactic expression to generalise over does not contain any recursive calls to its definition:

   $$n \notin tactics(t_{gen}).$$

The first two preconditions are what would be expected. The latter two, however, are to ensure that when X is instantiated with t_{gen} it forms a well-formed tactic that still evaluates. If there are any uninstantiated tactic variables, then evaluation will not succeed. Furthermore, if there are any recursive calls to the body of the tactic, these will be missing the new parameter and thus the tactic will not be well-formed.

TRANSFORMATION RULES This refactoring can be performed as a combination of other, simpler refactorings, most of which I have already shown:

1. Copy the tactic n and use n_{gen} as the new name. This refactoring is correct since Precondition 1 is exactly the precondition for *copy have*.
2. Swap the order of the two tactics, using the *swap statements* refactoring. I now have n_{gen} above n. The preconditions to this refactoring are satisfied as I know that n_{gen} does not contain any references to n, as a virtue of its origin from the *copy have* refactoring.
3. Now, I perform the generalisation on n_{gen}. This new refactoring, called *generalise unused tactic* is described below. The refactoring takes a fresh variable X_{gen} and a tactic expression to generalise t_{gen}, and will replace the tactic expression t_{gen} with the variable. The crucial precondition for this refactoring is that it must not be used in the proof. This fact ensures that the refactoring will not break the proof.
4. Finally, I perform tactic substitution to replace the body of n with an instance of n_{gen}. I perform this as a refactoring called *replace tactic definition*, which is also retailed below.

I write this transformation as follows:

$$\frac{prf \xrightarrow{copyhave} prf' \xrightarrow{swapstatements} prf'' \xrightarrow{generaliseunused} prf''' \xrightarrow{replacetacdef} prf_{gen}}{prf \xrightarrow{generalisetac} prf_{gen}} \text{(GT-PRF)}$$

GENERALISE UNUSED TACTIC Given parameters n: the name of the tactic to generalise; t_{gen}: the name of the expression to generalise; and, X_{gen}: the name of the new tactic variable. This refactoring is an instance of the modifier pattern, so I just give the modification rule:

$$\frac{t' = t[t_{gen} := X_{gen}]}{\begin{array}{l}\textbf{tac}\ n(\overline{X})\ :=\ t \\ stmts\end{array} \xrightarrow{generaliseunused} \begin{array}{l}\textbf{tac}\ n(\overline{X}, X_{gen})\ :=\ t' \\ stmts\end{array}} \text{(GUT-MOD)}$$

The precondition for this refactoring is that the new tactic variable is fresh and that the tactic is not used in the proof *prf*. The first condition is required to prove the well-formedness condition for the tactic (as part of the evaluation rule B-PRF-TAC). The latter condition is to ensure that the closure under environment extension theorem can be used to ensure that the remaining statements will still evaluate successfully.

REPLACE TACTIC DEFINITION Given a tactic name n and a tactic expression t', this refactoring will replace the body of a tactic with the expression t'. The transformation is a simple modification:

$$\frac{}{\begin{array}{l}\textbf{tac}\ n(\overline{X})\ :=\ t \\ stmts\end{array} \xrightarrow{replacetacdef} \begin{array}{l}\textbf{tac}\ n(\overline{X})\ :=\ t' \\ stmts\end{array}} \text{(RTD-MOD)}$$

This refactoring is semantics preserving if the following two preconditions hold:

1. The tactic variables in t' must be a subset of those in the definition:

$$vars(t') \subseteq \overline{X}$$

2. The new tactic must be equivalent to the old. That is to say, for an arbitrary list of goals, they both evaluate to return identical remaining goals.

CORRECTNESS I will not detail the correctness proof of this refactoring as it follows the same approach as other composite refactorings, like *rename have*.

9.18.1 *A variation on tactic generalisation*

Instead of creating a new tactic, such as *tnamegen* from *tname*:

```
proof
  tac tname := t1 ; t2 ; t3
  ...
  tac newtac := ... tname ...
qed
```

as described above, an alternative approach would be to generalise *tname* itself then *instantiate* the new parameter in any references to that tactic, resulting in a proof as below:

```
proof
  tac tname(X) := t1 ; X ; t3
  ...
  tac newtac := ... tname(t2) ...
qed
```

Happily, this is possible by further composing the version of *generalise tactic* described above, which results in a theory like

```
proof
  tac tnamegen(X) := t1 ; X ; t3
  tac tname := tnamegen(t2)
  ...
  tac newtac := ... tname ...
qed
```

with tactic substitutions to get a theory like:

```
proof
  tac tnamegen(X) := t1 ; X ; t3
  tac tname := tnamegen(t2)
  ...
  tac newtac := ... tnamegen(t2) ...
qed
```

Then, since *tname* is no longer used, it can be deleted and *tnamegen* can be renamed.

9.18.2 *Improving swap statements*

Building on the above observation, I can make the *move tactic up* instance of the *swap statements* refactoring more generally applicable. The idea is that if I have two statements:

```
have l: γ
  prf
tac mytac := t1 ;  lem l ; t2
...
show γ by mytac
```

then I can use this second form of *generalise tactic* on *mytac* with the expression *lem l* and then perform the move:

```
tac mytac(X) := t1 ;  X ; t2
have l: γ
  prf
...
show γ by mytac(lem l)
```

In this way, any defined tactics can be moved to the top of a proof block. This observation is important for the definition of the *local to global* refactoring in the next chapter.

9.19 NESTED PROOF REFACTORING

In the final refactoring of this chapter, I consider how a *proof refactoring* can be connected to a refactoring of a *lemma*:

```
lemma n: γ
  prf
```

This is straightforward if the refactoring I wish to perform is on the 'top-level' proof. If it is not:

```
lemma n: γ
proof
  ...
  show γ1
  proof
    ...
    show γ2
      prf
    ...
  qed
  ...
qed
```

then I can use the *nested proof refactoring* 'refactoring' to operate on it.

PARAMETERS This refactoring has the following parameters:

- A proof block *prf* to start with.
- The proof block to 'find' and refactor: prf_{find}.
- A refactoring $\xrightarrow{refac}$ to perform on prf_{find}.

PRECONDITIONS There are no explicit preconditions for this refactoring save the existence of prf_{find}. Of course, the supplied refactoring will not execute if its preconditions are not satisfied.

TRANSFORMATION RULES AND CORRECTNESS I only show a couple of rules and do not prove the correctness of this refactoring as it is relatively straightforward.

$$\frac{stmts \xrightarrow{nestedrefac} stmts'}{\begin{array}{ll}\textsf{proof}(t) & \xrightarrow{nestedrefac} \quad \textsf{proof}(t) \\ stmts & \quad stmts' \\ \textsf{qed} & \quad \textsf{qed}\end{array}} \quad \text{(NPR-Prf)}$$

$$\frac{prf = prf_{find} \qquad prf \xrightarrow{refac} prf'}{\textsf{show}\ \gamma\ prf\ stmts \xrightarrow{nestedrefac} \textsf{show}\ \gamma\ prf'\ stmts} \quad \text{(NPR-Show1)}$$

$$\frac{prf \neq prf_{find} \qquad prf \xrightarrow{nestedrefac} prf' \qquad stmts \xrightarrow{nestedrefac} stmts'}{\textsf{show}\ \gamma\ prf\ stmts \xrightarrow{nestedrefac} \textsf{show}\ \gamma\ prf'\ stmts'} \quad \text{(NPR-Show2)}$$

9.20 SUMMARY

In this chapter, I introduced formal definitions for refactoring and then over twenty proof refactorings. The refactorings ranged from simple transformations like *copy have* to sophisticated composite refactorings like *generalise tactic*. Each of these proof refactorings made localised changes to a particular proof block. Furthermore, each of the refactorings was proved to preserve semantics with respect to a given goal and a proof environment.

I have shown that refactorings can be combined in simple ways: sequentially, where one refactoring is performed then the second is applied to the result; and, exhaustively, where one refactoring is applied in as many places as possible, like the tactic substitution refactorings. Finally, I have identified some patterns of refactorings that occur frequently and developed a proof technique to simplify verification of correctness.

In the next chapter, I take these ideas further: defining refactorings that have a wider scope: a whole theory or even a whole proof document.

10

REFACTORING THEORIES

10.1 INTRODUCTION

In this chapter, I expand the scope of a refactoring to theories. First, in the next section I confine myself to refactorings that operate solely in a single theory. Then, in Section 10.3, I define several refactorings whose scope is (potentially) the whole proof document. Finally, I conclude with some discussion and a comparison with related work in Section 10.4.

These refactorings are defined using the syntax for proof documents, which is given in Figure 5.3.1 on page 69 and proved correct using the semantics from Figure 5.11 on page 73 and Section 5.3.4 on page 76.

10.2 SINGLE THEORY REFACTORINGS

Recall, from the previous chapter, that a *single theory refactoring* is a transformation $\xrightarrow{theory}$, where if:

$$thy \xrightarrow{theory} thy' \quad \text{and if} \quad \mathcal{D} \vdash theory \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \quad \text{and} \quad \mathcal{D} \vdash theory' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

then

$$\forall n.\ \mathcal{L}(n) = (visibility, \gamma, s) \implies \mathcal{L}'(n) = (visibility, \gamma, s')$$

and, for tactic environments:

$$\forall n.\ \mathcal{T}(n) = (visibility, \overline{X}, t) \implies \mathcal{T}'(n) = (visibility, \overline{X}, t).$$

That is to say:

- The proof environment contains at least the same names.
- The same goals are proved and tactic definitions exist.

While this sounds very restrictive, there are a few refactorings that fit into this category, such as *copy lemma* and *local to global*, which takes a **have** statement and turns it into a private lemma. These refactorings are detailed in the following sections. First, however, I will motivate and describe the proof technique used to show that these refactorings preserve semantics.

10.2.1 *Correctness preliminaries*

The top-level evaluation rule for a theory is the following:

$$\frac{\mathcal{D} \vdash \textbf{theory}\ name\ imports\ thyitems\ \textbf{end} : \langle \mathcal{T}', \mathcal{L}' \rangle \quad \mathcal{D} \vdash imports \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\mathcal{D} \vdash \textbf{theory}\ name\ imports\ thyitems\ \textbf{end} \Downarrow \langle \mathcal{T} \lhd \mathcal{T}_i, \mathcal{L} \lhd \mathcal{L}_i \rangle}$$

which states that a theory will evaluate to the environment $(\mathcal{T} \lhd \mathcal{T}_i, \mathcal{L} \lhd \mathcal{L}_i)$ if:

1. The theory is well-formed:

   $$\mathcal{D} \vdash \textbf{theory}\ name\ imports\ thyitems\ \textbf{end} : \langle \mathcal{T}', \mathcal{L}' \rangle$$

   where the 'top-level' well-formedness rule is:

   $$\frac{\mathcal{D} \vdash imports : \langle \mathcal{T}_i, \mathcal{L}_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle}{\mathcal{D} \vdash \textbf{theory}\ name\ imports\ thyitems\ \textbf{end} : \langle \mathcal{T}_i \lhd \mathcal{T}, \mathcal{L}_i \lhd \mathcal{L} \rangle}$$

2. The imports evaluate successfully:

   $$\mathcal{D} \vdash imports \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle$$

3. The contained *thyitems* evaluate successfully under the environment constructed by the imports:

   $$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

A characteristic of a single theory refactoring is that the 'header' information is not modified, only the *thyitems*. This means that the same imported environment is constructed before and after the refactoring. Thus, to show the correctness property for a theory, I only need to show it for the theory items.

Furthermore, because theories have split evaluation: well-formedness checking followed by proof checking, semantics preservation is a two-part process. This split evaluation model means that any proof of semantics preservation must show that the refactored theory is still well-formed and then that the proof checking process constructs a similar environment. Note the following two observations:

1. The proof environment constructed by well-formedness checking always contains an identity proof. This means the required relationship between the old and new, refactored environment is exactly the subset relation.

2. The proof environment constructed by well-formedness checking is identical to that constructed by the full proof checking process except that the hiproofs of each lemma are different.

Putting these two facts together means that if

$$thyitems \xrightarrow{theory} thyitems'$$

and

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}_{wf}, \mathcal{L}_{wf} \rangle \quad \text{and} \quad (\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

then I only need to show that the following four properties hold:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}'_{wf}, \mathcal{L}'_{wf} \rangle \tag{10.1}$$

$$\mathcal{T}_{wf} \subseteq \mathcal{T}'_{wf} \tag{10.2}$$

$$\mathcal{L}_{wf} \subseteq \mathcal{L}'_{wf} \tag{10.3}$$

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle \tag{10.4}$$

for some $\mathcal{T}'$ and $\mathcal{L}'$. That is, the refactored theory items are still well-formed and the constructed environments from well-formedness checking contain at least the same items as before the refactoring. Finally, I only need to show that the refactored theory items evaluate successfully.

10.2.2 *A proof refactoring: change proof*

As a first example, I demonstrate that any proof refactoring is also a *theory refactoring*. This follows the intuition since refactoring a proof preserves evaluation and does not modify the goal that it is applied to; the only potential side effect is that the resulting hiproof may change. This, of course, is acceptable for a single theory refactoring.

I demonstrate the transformation rules and correctness argument in detail for this simple example to give a good feel for the proof technique.

PARAMETERS The refactoring takes the following parameters:

- A theory to refactor, *theory*.
- The name of the lemma to apply it to, *lem*.
- A refactoring that one would wish to apply, $\xrightarrow{prfrefac}$.

PRECONDITIONS The preconditions for this refactoring are that the theory evaluates successfully, the lemma *lem* exists in the theory, and that all lemmas are *gap-free*. These are preconditions that will be required of all refactorings in this chapter.

TRANSFORMATION RULES The full set of transformation rules is as follows:

$$\frac{prf \xrightarrow{prfrefac} prf'}{\begin{array}{lcl} thyitems & \xrightarrow{changeproof} & thyitems \\ vis\ \textsf{lemma}\ lem : \gamma & & vis\ \textsf{lemma}\ lem : \gamma \\ prf & & prf' \end{array}} \tag{CP-Lem1}$$

$$\frac{n \neq lem \qquad thyitems \xrightarrow{changeproof} thyitems'}{\begin{array}{lll} thyitems & \xrightarrow{changeproof} & thyitems' \\ vis\ \textsf{lemma}\ n : \gamma & & vis\ \textsf{lemma}\ n : \gamma \\ prf & & prf \end{array}} \quad \text{(CP-Lem2)}$$

$$\frac{thyitems \xrightarrow{changeproof} thyitems'}{\begin{array}{lll} thyitems & \xrightarrow{changeproof} & thyitems' \\ vis\ \textsf{tac}\ n(\overline{X}) := t & & vis\ \textsf{tac}\ n(\overline{X}) := t \end{array}} \quad \text{(CP-Tac)}$$

$$\frac{thyitems \xrightarrow{changeproof} thyitems'}{\begin{array}{lll} thyitems & \xrightarrow{changeproof} & thyitems' \\ vis\ \textsf{hitac}\ n(\overline{X}) := t & & vis\ \textsf{hitac}\ n(\overline{X}) := t \end{array}} \quad \text{(CP-Hitac)}$$

Note that I do not have a transformation rule for a **begin** statement. The reason for this is because the transformation works backwards in a search for the lemma to refactor and it assumes that it definitely exists in the theory so will be reached and the refactoring will be complete. This style of transformation can be described as a *modify after* rule.

CORRECTNESS For this refactoring, I can prove that the proof environments constructed by the well-formedness checks are actually identical:

Theorem 34 (Equality of well-formedness checking for change proof)**.** *If, for a given imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$ *and theory items thyitems:*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle \quad and \quad thyitems \xrightarrow{changeproof} thyitems'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}, \mathcal{L} \rangle$$

Proof. The proof is an induction on the structure of the transformation rules. The 'base' case is:

CP-Lem1. This is the rule that modifies the proof. From the well-formedness assumption of the unchanged theory items, we know that the following is a valid derivation:

$$\frac{\begin{array}{c}\vdots\\ (\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle\end{array} \qquad lem \notin names(thyitems) \qquad \begin{array}{c}\vdots\\ (\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash prf\end{array}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems\ vis\ \textsf{lemma}\ lem : \gamma\ prf : \langle \mathcal{T}, \mathcal{L}[lem \mapsto (vis, \gamma, id)] \rangle}$$

and then need to show that

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle} \quad lem \notin names(thyitems) \quad \frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash prf'}$$
$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \; \textit{vis} \; \textsf{lemma} \; lem : \gamma \; prf' : \langle \mathcal{T}, \mathcal{L}[lem \mapsto (vis, \gamma, id)] \rangle$$

which simply amounts to showing that

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash prf'}$$

which is a direct consequence of the correctness of the refactoring $\xrightarrow{prfrefac}$.

There are then three 'step' cases. I only show one as the rest are identical:

- CP-Tac. On the left of this transformation rule, we know that the following derivation is valid:

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle} \quad \frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash t}$$
$$\frac{\mathrm{n} \notin names(thyitems) \qquad vars(t) \subseteq \overline{X}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \; \textit{vis} \; \textsf{tac} \; \mathrm{n}(\overline{X}) := t : \langle \mathcal{T}[\mathrm{n} \mapsto (vis, \overline{X}, t)], \mathcal{L} \rangle}$$

  and need to show that the theory items transformed by CP-Tac has the following derivation:

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}, \mathcal{L} \rangle} \quad \frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash t}$$
$$\frac{\mathrm{n} \notin names(thyitems) \qquad vars(t) \subseteq \overline{X}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' \; \textit{vis} \; \textsf{tac} \; \mathrm{n}(\overline{X}) := t : \langle \mathcal{T}[\mathrm{n} \mapsto (vis, \overline{X}, t)], \mathcal{L} \rangle}$$

  which is straightforward since $(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}, \mathcal{L} \rangle$ by the induction hypothesis.

□

Secondly, to show correctness of this refactoring I need to show that evaluation still occurs:

Theorem 35 (Evaluation correctness for change proof). *If, for a given imported environment $(\mathcal{T}_i, \mathcal{L}_i)$ and theory items thyitems I have:*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{and} \quad thyitems \xrightarrow{changeproof} thyitems'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

for some $\mathcal{T}'$, $\mathcal{L}'$.

Proof. Again, this is an induction on the transformation rules. It has an identical structure to the previous theorem, using the correctness property of the supplied proof refactoring to show that lemma evaluation still succeeds. □

Note that this refactoring and the proof technique are the same style as the modifier transformations from the previous chapter. This technique suffices for most of the single theory refactorings in this chapter. This example also illustrates that *strengthening* the theorem is useful to prove correctness of the refactoring.

10.2.3 *Copy an item*

Similar to the proof refactoring equivalent, one can copy a lemma or a tactic. In this section, I describe the *copy lemma* refactoring.

PARAMETERS The following parameters are required by this refactoring:

- A theory to refactor *theory*, or rather a list of theory items, *thyitems*.
- The name of the lemma to apply it to, *l*.
- The name for the copied lemma, *lnew*.

PRECONDITIONS Just like the proof refactoring equivalent, this refactoring must ensure that the new lemma name is fresh. For a theory, this means:

1. There are no other theory items with that name:

   $$lnew \notin names(thyitems)$$

2. It also does not appear as the name for any locally defined tactic or lemma *after* the new lemma position. If the list of theory items is:

   $thyitem_1$
   ...
   lemma $l : \gamma$
   prf
   ...
   $thyitem_n$

   then the property that, for every **lemma** (whose proof is *prf*) in **lemma** $\mathtt{l} \ldots thyitem_n$

   $$lnew \notin localnames(prf)$$

   must hold.

3. Finally, the lemma name $\mathtt{l}$ must not occur inside its proof. That is:

   $$\mathtt{l} \notin localnames(prf_{\mathtt{l}})$$

TRANSFORMATION RULES I give the full set of transformation rules for this refactoring:

$$\frac{}{\begin{array}{ll} \mathit{thyitems} & \mathit{thyitems} \\ \mathit{vis}\ \textsf{lemma}\ l : \gamma\ \mathit{prf} & \mathit{vis}\ \textsf{lemma}\ l : \gamma\ \mathit{prf} \\ & \mathit{vis}\ \textsf{lemma}\ \mathit{lnew} : \gamma\ \mathit{prf} \end{array}}\ \xrightarrow{\mathit{copylemma}} \quad \text{(CL-Lem1)}$$

$$\frac{\mathit{thyitems} \xrightarrow{\mathit{copylemma}} \mathit{thyitems}'}{\begin{array}{ll} \mathit{thyitems} & \xrightarrow{\mathit{copylemma}}\ \mathit{thyitems}' \\ \mathit{vis}\ \textsf{lemma}\ n : \gamma\ \mathit{prf} & \mathit{vis}\ \textsf{lemma}\ n : \gamma\ \mathit{prf} \end{array}} \quad \text{(CL-Lem2)}$$

$$\frac{\mathit{thyitems} \xrightarrow{\mathit{copylemma}} \mathit{thyitems}'}{\begin{array}{ll} \mathit{thyitems} & \xrightarrow{\mathit{copylemma}}\ \mathit{thyitems}' \\ \mathit{vis}\ \textsf{tac}\ n(\overline{X})\ :=\ t & \mathit{vis}\ \textsf{tac}\ n(\overline{X})\ :=\ t \end{array}} \quad \text{(CL-Tac)}$$

$$\frac{\mathit{thyitems} \xrightarrow{\mathit{copylemma}} \mathit{thyitems}'}{\begin{array}{ll} \mathit{thyitems} & \xrightarrow{\mathit{copylemma}}\ \mathit{thyitems}' \\ \mathit{vis}\ \textsf{hitac}\ n(\overline{X})\ :=\ t & \mathit{vis}\ \textsf{hitac}\ n(\overline{X})\ :=\ t \end{array}} \quad \text{(CL-Hitac)}$$

CORRECTNESS The transformation rules for this refactoring follows a correctness argument similar to the previous. This time, however, some more work is required to ensure the preconditions for the well-formedness checking rules for tactics and lemmas are still satisfied. To help do this, I strengthen the induction hypothesis by proving a more specific theorem about the resultant proof environments:

Theorem 36 (Correctness of well-formedness checking for copy lemma)**.** *If, for a given imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$ *and theory items thyitems:*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{and} \quad \mathit{thyitems} \xrightarrow{\mathit{copylemma}} \mathit{thyitems}'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}' : \langle \mathcal{T}, \mathcal{L}[\mathit{lnew} \mapsto (\mathit{vis}, \gamma, \mathit{id})] \rangle.$$

That is, the refactored theory constructs identical environments, except that the lemma environment has been extended to include one more item.

Proof. This proof is given in Section C.7 of Appendix C. □

```
public lemma comm P ∧ Q ⇒ Q ∧ P
proof(impl)
  tac t1 := conjE ; ax
  have q: P ∧ Q ⊢ Q
    by t1
  have p: P ∧ Q ⊢ P
    by t1
  from q p show P ∧ Q ⊢ Q ∧ P by conjI
qed
```

Listing 10.1: Lemma with local lemmas and tactics to extract

10.2.4 *Swapping items*

This refactoring is very similar to its equivalent for Hiscript proofs. It is possible to:

- Move a tactic definition above another tactic definition;
- Move a tactic definition above a lemma;
- Move a lemma above another lemma;
- Move a lemma above a tactic definition.

For each of these, the precondition is similar: the item being moved upwards must not rely on the item that it is being moved above. Because of the similarity to the proof refactoring equivalent, I do not present any details of this refactoring.

10.2.5 *Local to global*

In the *local to global* refactoring, a local statement, such as a **have** or a **tac** statement is granted the status of a *private* **lemma** or (global) **tac**, respectively.

In order to define this refactoring, I take advantage of some proof refactorings to move the statement to extract as a lemma or tactic to the top of the proof block it is currently in. To illustrate why this is important, consider the lemma in Listing 10.1. I wish to apply *local to global* on the tactic *t1*. This would be successful as long as the name is not used further down the theory, which is checked with exactly the preconditions as *copy item* above. The resulting theory fragment would look like Listing 10.2 as expected.

However, imagine instead extracting the local lemma q, which would result in a theory fragment like that shown in Listing 10.3. This theory is no longer well-formed. The local tactic *t1* that was used to prove q within the lemma *comm* is no longer in the environment.

Thus, this refactoring requires the statement to extract to be at the top of the proof block. This is not always possible for a local lemma as I have just shown, but for tactics, it can always be done. To see why, consider the following variation on the example in Listing 10.4 The general form of *swapping statements* described in Section 9.18.2 can be used to move *t2* to the top of the proof block, as shown in Listing 10.5, where it

```
private tac t1 := conjE ; ax

public lemma comm P ∧ Q ⇒ Q ∧ P
proof(impl)
  have q: P ∧ Q ⊢ Q
    by t1
  have p: P ∧ Q ⊢ P
    by t1
  from q p show P ∧ Q ⊢ Q ∧ P by conjI
qed
```

Listing 10.2: Theory fragment with global t1

```
private lemma q: P ∧ Q ⊢ Q
  by t1

public lemma comm P ∧ Q ⇒ Q ∧ P
proof(impl)
  tac t1 := conjE ; ax
  have p: P ∧ Q ⊢ P
    by t1
  from q p show P ∧ Q ⊢ Q ∧ P by conjI
qed
```

Listing 10.3: Badly formed theory after extracting the lemma q

```
public lemma comm P ∧ Q ⇒ Q ∧ P
proof(impl)
  tac t1 := conjE ; ax
  have q: P ∧ Q ⊢ Q
    by t1
  have p: P ∧ Q ⊢ P
    by t1
   tac t2 := conjI ; q ⊗ p
  show P ∧ Q ⊢ Q ∧ P by t2
qed
```

Listing 10.4: A variation on the example where *t2* is extracted

```
public lemma comm P ∧ Q ⇒ Q ∧ P
proof(impl)
  tac t2(X,Y) := conjI ; Y ⊗ X
  tac t1 := conjE ; ax
  have q: P ∧ Q ⊢ Q
    by t1
  have p: P ∧ Q ⊢ P
    by t1
  show P ∧ Q ⊢ Q ∧ P by t2(p,q)
qed
```

Listing 10.5: Using swap statements and generalisation to move t2

can be refactored. I now describe the parameters, preconditions, transformation rules and correctness of the tactic version of this refactoring.

PARAMETERS This refactoring takes three parameters:

- The theory (and associated theory items to refactor), *theory* (*thyitems*).
- The name of the lemma that contains the local tactic to extract, *lem*.
- The name of the local tactic, *tname*.

PRECONDITIONS There is one precondition for this refactoring. The name of the local tactic must not clash with any names defined later in the theory items (global or local). I split this into three separate cases:

1. The name of local tactic is not the same as the lemma: $tname \neq lem$.
2. There are no other theory items with that name:

   $$tname \notin names(thyitems)$$

3. It also does not appear as the name for any locally defined tactic or lemma *after* the lemma position. If the list of theory items is:

   $thyitem_1$
   ...
   lemma $lem : \gamma$
   prf
   $thyitem_i$
   ...
   $thyitem_n$

   then the property that, for every **lemma** (whose proof is prf) in $thyitem_i \ldots thyitem_n$

   $$tname \notin localnames(prf)$$

   must hold.

TRANSFORMATION RULES There are two rules:

$$\frac{}{\begin{array}{l}\mathit{thyitems}\\ \mathit{vis}\ \textsf{lemma}\ \mathit{lem}: \gamma\\ \textsf{proof}(t)\\ \quad \textsf{tac}\ \mathit{tname}(\overline{X}) := t\\ \quad \mathit{stmts}\\ \textsf{qed}\end{array} \xrightarrow{\mathit{localglobal}} \begin{array}{l}\mathit{thyitems}\\ \textsf{private tac}\ \mathit{tname}(\overline{X}) := t\\ \\ \mathit{vis}\ \textsf{lemma}\ \mathit{lem}: \gamma\\ \textsf{proof}(t)\\ \quad \mathit{stmts}\\ \textsf{qed}\end{array}} \quad \text{(LG-Mod)}$$

$$\frac{\mathit{thyitems} \xrightarrow{\mathit{localglobal}} \mathit{thyitems}'}{\begin{array}{ll}\mathit{thyitems} & \xrightarrow{\mathit{localglobal}} \mathit{thyitems}'\\ \mathit{thyitem} & \mathit{thyitem}\end{array}} \quad \text{(LG-ThyItem)}$$

The transformation rules recurse through the theory items (using LG-ThyItem) until the lemma to refactor has been found. Then LG-Mod will perform the extraction step, placing the new tac above the lemma and removing the local definition.

CORRECTNESS I will show the first part of the correctness proof for this refactoring:

Theorem 37 (Correctness of well-formedness checking for local to global)**.** *If, for a given imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$ *and theory items thyitems*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{and} \quad \mathit{thyitems} \xrightarrow{\mathit{localglobal}} \mathit{thyitems}'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}' : \langle \mathcal{T}[\mathit{tname} \mapsto (\textbf{\textit{private}}, \overline{X}, t)], \mathcal{L} \rangle.$$

That is, the refactored theory constructs identical environments, except that the tactic environment has been extended to include the new global tactic.

Proof. The proof is detailed in Section C.8 of Appendix C. □

10.2.6 *Private to public*

This refactoring is an easy, precondition-free refactoring in Hiscript, with a modifier pattern with the obvious rule. It is applicable in general, since in Hiscript there is no likelihood of a name clash in the wider environment (of the proof document) since all names are prefixed by the theory name. I do not give any further details.

10.3 REFACTORING A PROOF DOCUMENT

Finally for this chapter, I specify a few refactorings that may modify a range of theories in a proof document. As discussed in Section 9.2, I consider three degrees of semantics preservation for proof document refactorings:

- *Well-formedness preserving*, where the only guarantee is that the refactoring still allows the proof document to evaluate.
- *Weak semantics preserving*, where at least the same lemmas are proved before and after the refactoring.
- *Strong semantics preserving*, which is weakly semantics preserving, but the lemmas are proved in the same theories as before.

In practice, though, it is often easier to prove a more precise preservation property about each refactoring, as a stronger induction hypothesis is often vital for the proof. These generic semantics preservation properties then follow.

10.3.1 *Single theory refactorings*

Recall from above that a single theory refactoring can only extend the proof environment or change the hiproof of the lemmas. Thus, performing a single theory refactoring will clearly be *strong semantics preserving*.

PARAMETERS This refactoring takes three parameters:

1. The theory map to refactor, *thys*, considered as a list of pairs of theory name and body.
2. A *single theory refactoring*, $\xrightarrow{str}$.
3. The name of the theory to refactor, n_{str}.

PRECONDITIONS There are no preconditions to this refactoring, except that the preconditions to $\xrightarrow{str}$ will hold.

TRANSFORMATION RULES This refactoring is performed by navigating through the theory map until the theory to refactor is found, then applying the single theory refactoring as defined by the following rules:

$$\frac{theory \xrightarrow{str} theory'}{\begin{array}{ll} thys & \xrightarrow{refactortheory} \quad thys \\ (n_{str}, theory) & \qquad\qquad\quad (n_{str}, theory') \end{array}} \quad \text{(STR-Mod)}$$

$$\frac{n \neq n_{str} \qquad thys \xrightarrow{refactortheory} thys'}{\begin{array}{ll} thys & \xrightarrow{refactortheory} \quad thys' \\ (n, theory) & \qquad\qquad\quad (n, theory) \end{array}} \quad \text{(STR-Thys)}$$

CORRECTNESS To prove that this refactoring is strongly semantics preserving, I prove a stronger theorem:

Theorem 38 (Correctness of single theory refactoring)**.** *Let* $thys \xrightarrow{refactortheory} thys'$, $\vdash thys \Downarrow \mathcal{E}$, *and* $\vdash thys' \Downarrow \mathcal{E}'$. *Then, for every theory name* $\mathfrak{n}$, *such that* $\mathcal{E}(\mathfrak{n}) = (\mathcal{T}, \mathcal{L})$ *and* $\mathcal{E}'(\mathfrak{n}) = (\mathcal{T}', \mathcal{L}')$ *I have:*

$$\mathcal{T} \subseteq \mathcal{T}'$$

$$\mathcal{L} \subseteq_s \mathcal{L}'$$

where $\subseteq_s$ *means the lemma environments are all equal except that the hiproof may differ and that* $\mathcal{L}'$ *may contain extra lemmas.*

Proof. The proof is an induction on the transformation rules. There are two cases:

- The base case: STR-MOD. By assumption, we know that the following derivation is valid:

$$\frac{\vdash thys \Downarrow \mathcal{E} \qquad thys \vdash theory \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash thys \ \boxed{(n_{str}, theory)} \Downarrow \mathcal{E}[n_{str} \mapsto (\mathcal{T}, \mathcal{L})]}$$

  and we need to show that the derivation is valid and that the theorem property holds:

$$\frac{\vdash thys \Downarrow \mathcal{E} \qquad thys \vdash theory' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle}{\vdash thys \ \boxed{(n_{str}, theory)} \Downarrow \mathcal{E}[n_{str} \mapsto (\mathcal{T}', \mathcal{L}')]}$$

  Now, we know it is a valid derivation using the assumption that $\vdash thys \Downarrow \mathcal{E}$ and that $\xrightarrow{str}$ is a refactoring (recall that a refactoring preserves evaluation). Thus, we just need to show that $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{L} \subseteq_s \mathcal{L}'$, but this is a direct consequence of the semantics preservation property of a single theory refactoring (from Section 9.2).

- The step case: STR-THYS. By assumption, we know that the following derivation is valid:

$$\frac{\vdash thys \Downarrow \mathcal{E} \qquad thys \vdash theory \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash thys \ \boxed{(n, theory)} \Downarrow \mathcal{E}[n \mapsto (\mathcal{T}, \mathcal{L})]}$$

  and we need to show that the following is valid:

$$\frac{\vdash thys' \Downarrow \mathcal{E}' \qquad thys' \vdash theory \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle}{\vdash thys \ \boxed{(n, theory)} \Downarrow \mathcal{E}'[n \mapsto (\mathcal{T}', \mathcal{L}')]}$$

  First, by the induction hypothesis, we know that $\vdash thys' \Downarrow \mathcal{E}'$. Furthermore, we also know that each member of $\mathcal{E}'$ contains at least the lemmas and tactics of $\mathcal{E}$. Now, we need to show that:

$$thys' \vdash theory \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

where $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{L} \subseteq_s \mathcal{L}'$. The main rule for evaluating a theory is THY-EVAL, thus we need to show that, if the following derivation is valid:

$$\frac{\begin{array}{c} \mathit{thys} \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} : \langle \mathcal{T}_{wf}, \mathcal{L}_{wf} \rangle \\ \mathit{thys} \vdash \mathit{imports} \Downarrow \langle \mathcal{T}_i, \mathcal{L}_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \end{array}}{\mathit{thys} \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} \Downarrow \langle \mathcal{T} \lhd \mathcal{T}_i, \mathcal{L} \lhd \mathcal{L}_i \rangle}$$

then the following is also valid

$$\frac{\begin{array}{c} \mathit{thys}' \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} : \langle \mathcal{T}'_{wf}, \mathcal{L}'_{wf} \rangle \\ \mathit{thys}' \vdash \mathit{imports} \Downarrow \langle \mathcal{T}'_i, \mathcal{L}'_i \rangle \qquad (\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle \end{array}}{\mathit{thys}' \vdash \textbf{theory}\ \mathit{name}\ \mathit{imports}\ \mathit{thyitems}\ \textbf{end} \Downarrow \langle \mathcal{T}' \lhd \mathcal{T}'_i, \mathcal{L}' \lhd \mathcal{L}'_i \rangle}$$

which further means showing that if the theory is well-formed under the original theory map, then it is also well-formed under the modified theory map. This part of the proof is similar to the evaluation part, which I now show.

To prove this derivation is valid, we show that:

1. $\mathcal{T}_i \subseteq \mathcal{T}'_i$;
2. $\mathcal{L}_i \subseteq \mathcal{L}'_i$;
3. $\mathcal{T} \subseteq \mathcal{T}'$;
4. and, $\mathcal{L} \subseteq \mathcal{L}'$.

For the imported environments, we need to use the relationship between $\vdash \mathit{thys} \Downarrow \mathcal{E}$ and $\vdash \mathit{thys}' \Downarrow \mathcal{E}'$. The rules for importing (IMPORT-1-EVAL and IMPORT-2-EVAL) construct the imported environment by a union of each individual import. Now, by the induction hypothesis about the relationship between $\mathcal{E}$ and $\mathcal{E}'$, we know that each environment satisfies the property. This means their union will also satisfy it.

Now, to show that

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

is valid, we use the environment extension theorem for theory items, Theorem 54 on page 221. This theorem states that if a list of theory items evaluates under an environment $(\mathcal{T}, \mathcal{L})$ then it will also evaluate against a well-formed environment that is larger $(\mathcal{T}', \mathcal{L}')$.

□

The next two refactorings have a similar style: recurse through the theory map until the appropriate theory is found, then use some specific rules to refactor it.

10.3.2 *Public to private*

This refactoring will change the visibility of a particular theory item in a given theory. In order for it to be behaviour preserving, the item cannot be used outside the theory in which it is defined. As a consequence, this refactoring has a precondition that is global throughout the proof document, but the action is localised. In order to represent this refactoring, it is split into two parts:

- Recurse through the theory map until the theory containing the theory item to modify is found.
- Recurse through that theory to find and then modify the item.

PARAMETERS The parameters for this refactoring are:

1. The proof document to refactor, *thys*.
2. The name of the theory that contains the item to modify, n_{pub}.
3. The name of the item to change to **private**, n.

PRECONDITIONS The precondition to this refactoring is that the item to make private is not used anywhere in the document except in its local theory. This can be expressed by saying that it is not in the minimal environment of any theory in the document except n_{pub}.

TRANSFORMATION RULES The rules operating on the proof document are

$$\frac{theory_{pub} \xrightarrow{publicprivate} theory'_{pub}}{\begin{array}{ll} thys & \xrightarrow{publicprivate} \quad thys \\ (\mathfrak{n}_{pub}, theory_{pub}) & \quad\quad\quad\quad\quad (\mathfrak{n}_{pub}, theory'_{pub}) \end{array}} \quad \text{(PP-DocMod)}$$

$$\frac{\mathfrak{n} \neq \mathfrak{n}_{pub} \qquad thys \xrightarrow{publicprivate} thys'}{\begin{array}{ll} thys & \xrightarrow{publicprivate} \quad thys' \\ (\mathfrak{n}, theory) & \quad\quad\quad\quad\quad (\mathfrak{n}, theory) \end{array}} \quad \text{(PP-Thys)}$$

and the rules operating on the theory are:

$$\frac{thyitems \xrightarrow{publicprivate} thyitems'}{\begin{array}{ll} \textbf{theory } \mathfrak{n}_{pub} & \xrightarrow{publicprivate} \quad \textbf{theory } \mathfrak{n}_{pub} \\ imports & \quad\quad\quad\quad\quad imports \\ \textbf{begin} & \quad\quad\quad\quad\quad \textbf{begin} \\ \quad thyitems & \quad\quad\quad\quad\quad\quad thyitems' \\ \textbf{qed} & \quad\quad\quad\quad\quad \textbf{qed} \end{array}} \quad \text{(PP-Thy)}$$

$$\frac{name(thyitem_{pub}) = \mathfrak{n}}{\begin{array}{lll} thyitems & \xrightarrow{publicprivate} & thyitems \\ \textbf{public}\ thyitem_{pub} & & \textbf{private}\ thyitem_{pub} \end{array}} \quad \text{(PP-ThyMod)}$$

$$\frac{name(thyitem) \neq \mathfrak{n} \qquad thyitems \xrightarrow{publicprivate} thyitems'}{\begin{array}{lll} thyitems & \xrightarrow{publicprivate} & thyitems' \\ thyitem & & thyitem \end{array}} \quad \text{(PP-ThyItem)}$$

CORRECTNESS This refactoring is *strongly semantics preserving*, which means that I must prove:

Theorem 39 (Correctness of public to private)**.** *If* $\vdash thys \Downarrow \mathcal{E}$ *and* $thys \xrightarrow{publicprivate} thys'$ *then* $\vdash thys' \Downarrow \mathcal{E}'$ *where*

$$\forall l \in \mathcal{L}.\ \mathcal{L}(l) = (\gamma, s) \rightarrow \exists l' \in \mathcal{L}'.\ \mathcal{L}'(l') = (\gamma, s')$$

In fact, I prove something stronger:

Theorem 40 (Strong correctness of public to private)**.** *If* $\vdash thys \Downarrow \mathcal{E}$ *and* $thys \xrightarrow{publicprivate} thys'$ *then* $\vdash thys' \Downarrow \mathcal{E}'$*. Then, for all theory names* $\mathfrak{n}$*, if* $\mathcal{E}(\mathfrak{n}) = (\mathcal{T}, \mathcal{L})$ *and* $\mathcal{E}(\mathfrak{n})' = (\mathcal{T}', \mathcal{L}')$*, then:*

$$\mathcal{T}\big|_{vis} = \mathcal{T}'\big|_{vis}$$

and

$$\mathcal{L}\big|_{vis} = \mathcal{L}'\big|_{vis}$$

That is, if the visibilities are removed, then the environments are identical.

Proof. The proof proceeds by induction on the transformation rules for a document. The proof is similar to the previous refactoring, though, instead of relying on the correctness of the supplied refactoring, we argue by induction on the *theory item* transformation rules: that the transformation preserves the correctness property. This proof is straightforward. Again, the tricky part is for the rule PP-Thys, where we need to show that if

$$thys \vdash theory \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

then

$$thys' \vdash theory \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

is valid and that $(\mathcal{T}', \mathcal{L}')$ satisfies the property and it is proved in a similar way to before. □

10.3.3 *Delete unused item*

This refactoring performs a similar operation to the *delete unused have statement* refactoring given in Section 9.4. Rather than checking that the item is unused throughout the proof document, I make the assumption that it is **private** then ensure that it is unused in the theory in which it is defined.

PARAMETERS This refactoring also takes three parameters:

1. The proof document to refactor, *thys*.
2. The name of the theory that contains the item to delete, n_{del}. The theory itself I will refer to as $theory_{del}$.
3. The name of the item to delete, n.

PRECONDITIONS The one precondition is that the item to delete is not used in $theory_{del}$. That is:

$$n_{del} \notin (lemmas(theory_{del}) \cup tactics(theory_{del}))$$

where the functions *lemmas* and *tactics* are extended from the *prf* equivalents to return the set of all used lemmas and tactics.

TRANSFORMATION RULES The transformation rules for this refactoring are similar to the previous:

$$\frac{theory_{del} \xrightarrow{deleteitem} theory'_{del}}{\begin{array}{ll} thys & \xrightarrow{deleteitem} \quad thys \\ (\mathfrak{n}_{del}, theory_{del}) & \quad (\mathfrak{n}_{pub}, theory'_{del}) \end{array}} \quad \text{(DI-DocMod)}$$

$$\frac{\mathfrak{n} \neq \mathfrak{n}_{del} \qquad thys \xrightarrow{deleteitem} thys'}{\begin{array}{ll} thys & \xrightarrow{deleteitem} \quad thys' \\ (\mathfrak{n}, theory) & \quad (\mathfrak{n}, theory) \end{array}} \quad \text{(DI-Thys)}$$

and the rules operating on the theory are:

$$\frac{thyitems \xrightarrow{deleteitem} thyitems'}{\begin{array}{ll} \textbf{theory } \mathfrak{n}_{del} & \xrightarrow{deleteitem} \quad \textbf{theory } \mathfrak{n}_{del} \\ imports & \quad imports \\ \textbf{begin} & \quad \textbf{begin} \\ \quad thyitems & \quad\quad thyitems' \\ \textbf{qed} & \quad \textbf{qed} \end{array}} \quad \text{(DI-Thy)}$$

$$\frac{name(thyitem_{del}) = \mathfrak{n}}{\begin{array}{ll} thyitems & \xrightarrow{deleteitem} \; thyitems \\ thyitem_{del} & \end{array}} \quad \text{(DI-ThyMod)}$$

$$\frac{name(thyitem) \neq \mathfrak{n} \qquad thyitems \xrightarrow{deleteitem} thyitems'}{\begin{array}{ll} thyitems & \xrightarrow{deleteitem} \; thyitems' \\ thyitem & \qquad\quad thyitem \end{array}} \quad \text{(DI-ThyItem)}$$

CORRECTNESS To show that this refactoring is well-formedness preserving, one needs to assume a stronger induction hypothesis. For this correctness property, I prove that the constructed theorem environment map contains identical proof environments, except that they do not contain an item with the name of the deleted item, $\mathfrak{n}$. I do not detail the proof here.

10.3.4 *Substitution in a theory*

Just as substitution inside a tactic could be extended to a declarative proof, I extend the lemma and tactic substitution refactoring to the level of theories. I write

$$theory[lem\ l := t]$$

to mean the substitution of a tactic expression t for a lemma *lem l* in *theory*. This means substituting it inside all the lemmas and tactic definitions.

PARAMETERS This refactoring requires three parameters:

1. The theory to refactor, *theory*.
2. The name of the lemma to substitute, $\mathfrak{l}$.
3. The tactic expression to substitute it for, $t_{\mathfrak{l}}$.

PRECONDITIONS The precondition for this refactoring is that the lemma and tactic expression behave identically over any input goal:

$$\forall \gamma.\ \langle \gamma, lem\ l \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g, s \rangle \rightarrow \langle \gamma, t_{\mathfrak{l}} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g, s' \rangle$$

TRANSFORMATION RULES The rules for transforming this theory are as follows:

$$\frac{thyitems \xrightarrow{theorysubstitute} thyitems'}{\begin{array}{ll} \textbf{theory} & \xrightarrow{theorysubstitute} \; \textbf{theory} \\ imports & \qquad\quad imports \\ \quad thyitems & \qquad\quad\ thyitems' \\ \textbf{qed} & \qquad\quad \textbf{qed} \end{array}} \quad \text{(TS-Thy)}$$

$$\frac{}{\textsf{begin} \xrightarrow{\textit{theorysubstitute}} \textsf{begin}} \quad \text{(TS-BEGIN)}$$

$$\frac{t' = t[\textit{lem}\ l := t_1] \qquad \textit{thyitems} \xrightarrow{\textit{theorysubstitute}} \textit{thyitems}'}{\begin{array}{ll}\textit{thyitems} & \xrightarrow{\textit{theorysubstitute}} \quad \textit{thyitems} \\ \textit{vis}\ \textsf{tac}\ \mathfrak{n}(\overline{X})\ := \ t & \textit{vis}\ \textsf{tac}\ \mathfrak{n}(\overline{X})\ := \ t'\end{array}} \quad \text{(TS-TAC)}$$

$$\frac{t' = t[\textit{lem}\ l := t_1] \qquad \textit{thyitems} \xrightarrow{\textit{theorysubstitute}} \textit{thyitems}'}{\begin{array}{ll}\textit{thyitems} & \xrightarrow{\textit{theorysubstitute}} \quad \textit{thyitems} \\ \textit{vis}\ \textsf{hitac}\ \mathfrak{n}(\overline{X})\ := \ t & \textit{vis}\ \textsf{hitac}\ \mathfrak{n}(\overline{X})\ := \ t'\end{array}} \quad \text{(TS-HITAC)}$$

$$\frac{\textit{prf}' = \textit{prf}[\textit{lem}\ l := t_1] \qquad \textit{thyitems} \xrightarrow{\textit{theorysubstitute}} \textit{thyitems}'}{\begin{array}{ll}\textit{thyitems} & \xrightarrow{\textit{theorysubstitute}} \quad \textit{thyitems} \\ \textit{vis}\ \textsf{lemma}\ \mathfrak{n} :\ \gamma\ \textit{prf} & \textit{vis}\ \textsf{lemma}\ \mathfrak{n} :\ \gamma\ \textit{prf}'\end{array}} \quad \text{(TS-LEMMA)}$$

CORRECTNESS This refactoring produces a proof environment that contains equivalent tactic definitions and lemmas with possibly different hiproofs:

Theorem 41 (Correctness of theory substitute)**.** *If, for some imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$*,* $\mathcal{D} \vdash \textit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$ *and if theory* $\xrightarrow{\textit{theorysubstitute}}$ *theory*$'$ *then*

$$\mathcal{D} \vdash \textit{theory}' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

where $\mathcal{L} =_s \mathcal{L}'$ *and* $\mathcal{T} =_t \mathcal{T}'$*.*

Proof. The proof of this theorem is similar to the previous proofs and appeals to the appropriate correctness properties of the appropriate tactic and proof substitution refactorings where necessary. □

A consequence of this refactoring is that the lemma l will no longer be in the minimal environment for *theory*. This is an important fact for composition of refactorings for the definition of *rename item*, described below.

10.3.5 *Rename item*

Renaming is a classic refactoring and it is only now that I have built up enough machinery to define it for a proof document. In this section, I will define the *rename lemma* version.

It is also important to note that there are two variants:

- The lemma is **private**. In this case, the renaming can be localised to the theory in which it is defined.
- The lemma is **public**. In this case, the renaming must be performed across all instances of it in the proof document.

I will describe the latter, more complicated refactoring here. As with proof refactoring, I define this refactoring in terms of a composition of *copy item*, *delete unused item*, and *lemma substitution*.

PARAMETERS The refactoring takes four parameters:

1. The proof document to refactor, *thys*.
2. The name of the theory in which the lemma to rename is proved, n_{rename}. I refer to the theory items in that theory as $thyitems_{rename}$.
3. The name of the lemma to rename, *lold*.
4. Finally, the new name for the lemma, *lnew*.

PRECONDITIONS The preconditions for this refactoring are exactly the same as for the *copy an item* refactoring described in Section 10.2.3:

1. There are no other theory items with that name:

   $$lnew \notin names(thyitems_{rename})$$

2. It also does not appear as the name for any locally defined tactic or lemma *after* the new lemma position. If the list of theory items is:

   $thyitem_1$
   ...
   lemma lold : γ
   prf
   ...
   $thyitem_n$

   then the property that, for every **lemma** (whose proof is *prf*) in **lemma** *lold* ... $thyitem_n$

   $$lnew \notin localnames(prf)$$

   must hold.

3. Finally, the lemma name *lold* must not occur inside its proof. That is:

   $$lold \notin localnames(prf_{lold})$$

TRANSFORMATION RULES This refactoring is defined in two steps:

$$\frac{thys \xrightarrow{renamelemma'} thys' \qquad thys' \xrightarrow{publicprivate(lemold)} thys'' \xrightarrow{deleteitem(lemold)} thys'''}{thys \xrightarrow{renamelemma} thys'''} \text{(RL-Top)}$$

where $\xrightarrow{renamelemma'}$ is defined by the rules:

$$\frac{\begin{array}{ll} thys & \\ (\mathfrak{n}_{rename}, theory) & \end{array} \quad theory \xrightarrow{copyitem} theory' \quad theory'' = theory'[lem\ lold := lem\ lnew]}{\begin{array}{ll} thys & \\ (\mathfrak{n}_{rename}, theory'') & \end{array}} \text{(RL-Mod)}$$

$$\frac{\mathfrak{n} \neq \mathfrak{n}_{rename} \quad \begin{array}{ll} thys & \\ (\mathfrak{n}, theory) & \end{array} \quad theory' = theory[lem\ lold := lem\ lnew] \quad thys \xrightarrow{renamelemma'} thys'}{\begin{array}{ll} thys & \\ (\mathfrak{n}, theory') & \end{array}} \text{(RL-Thys)}$$

The rule RL-Thys will perform substitution of the new lemma for the old name in any theories after the modified one. Then, once the theory containing the definition is reached, the rule RL-Mod will create a copy of the lemma, then substitutes any local references to the old lemma with a reference to the copied lemma.

At the top level of this refactoring, the rule RL-Top will first perform the transformation given by those rules, then it will make the old lemma private, then delete it. I indicate the parameters for these refactoring explicitly.

CORRECTNESS This refactoring is strongly semantics preserving:

Theorem 42 (Correctness of rename lemma)**.** *If* $\vdash thys \Downarrow \mathcal{E}$ *and* $thys \xrightarrow{renamelemma} thys'$ *then* $\vdash thys' \Downarrow \mathcal{E}'$ *where*

$$\forall l \in \mathcal{L}.\ \mathcal{L}(l) = (\gamma, s) \rightarrow \exists l' \in \mathcal{L}'.\ \mathcal{L}'(l') = (\gamma, s')$$

Proof. To prove this, we show that the preconditions for each refactoring are met, which then implies the correctness of *rename lemma*. First, the preconditions of *copy item* are directly matched by those of this refactoring. Secondly, since the lemma substitution is $theory[lem\ lold := lem\ lnew]$ and the lemmas are identical, the precondition for *theory substitute* is met. A consequence of that refactoring is that the substituted lemma, *lold*, is no longer in the minimal environment for that theory. This is precisely the precondition for the refactoring *public to private*. Finally, the preconditions of *delete unused item* are met by a consequence that *lold* is now a **private** lemma and by the fact that *lold* is not in the minimal environment for the theory in which it is defined (a consequence of *theory substitute*). □

10.3.6 *Delete theory*

The *delete theory* refactoring is *well-formedness preserving* only, since it actively removes a set of lemmas (inside a theory).

PARAMETERS This refactoring only takes two parameters:

- *thys*, the document to refactor.
- $\mathfrak{n}_{del}$, the name of the theory to delete.

PRECONDITIONS A theory can only be deleted if it is not imported by any theory:

$$\forall\ (\mathfrak{n}, \mathit{theory}) \in \mathit{thys}.\ \mathfrak{n}_{\mathit{del}} \notin \mathit{imports}(\mathit{theory})$$

TRANSFORMATION RULES This refactoring follows a modifier-style pattern at the level of theories:

$$\frac{}{\begin{array}{ll}\mathit{thys} & \xrightarrow{\mathit{deletetheory}} \quad \mathit{thys} \\ (\mathfrak{n}_{\mathit{del}}, \mathit{theory}_{\mathit{del}}) & \end{array}} \qquad \text{(DT-Mod)}$$

$$\frac{\mathfrak{n} \neq \mathfrak{n}_{\mathit{del}} \qquad \mathit{thys} \xrightarrow{\mathit{deletetheory}} \mathit{thys}'}{\begin{array}{lll}\mathit{thys} & \xrightarrow{\mathit{deletetheory}} & \mathit{thys}' \\ (\mathfrak{n}, \mathit{theory}) & & (\mathfrak{n}, \mathit{theory})\end{array}} \qquad \text{(DT-Thy)}$$

CORRECTNESS To show that this refactoring is correct:

Theorem 43 (Correctness of delete theory)**.** *If* $\mathit{thys} \vdash \mathcal{E}$ *and* $\mathit{thys} \xrightarrow{\mathit{deletetheory}} \mathit{thys}'$ *then* $\mathit{thys}' \vdash \mathcal{E}'$ *for some* $\mathcal{E}'$*.*

Proof. The proof is an induction on the transformation rules. The base case, for the rule DT-Mod, is trivial.

For the step case, we know

$$\frac{\mathfrak{n} \notin \mathit{thynames}(\mathit{thys}) \qquad \mathit{imports}(\mathit{theory}) \subseteq \mathit{thys} \qquad \vdash \mathit{thys} \Downarrow \mathcal{E} \qquad \mathit{thys} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}_{\mathfrak{n}}, \mathcal{L}_{\mathfrak{n}} \rangle}{\vdash \mathit{thys}\ (\mathfrak{n}, \mathit{theory}) \Downarrow \mathcal{E}[\mathfrak{n} \mapsto (\mathcal{T}_{\mathfrak{n}}, \mathcal{L}_{\mathfrak{n}})]}$$

and need to show that the refactored theories still evaluates. That is:

$$\frac{\mathfrak{n} \notin \mathit{thynames}(\mathit{thys}') \qquad \mathit{imports}(\mathit{theory}) \subseteq \mathit{thys}' \qquad \vdash \mathit{thys}' \Downarrow \mathcal{E}' \qquad \mathit{thys}' \vdash \mathit{theory} \Downarrow \langle \mathcal{T}'_{\mathfrak{n}}, \mathcal{L}'_{\mathfrak{n}} \rangle}{\vdash \mathit{thys}'\ (\mathfrak{n}, \mathit{theory}) \Downarrow \mathcal{E}'[\mathfrak{n} \mapsto (\mathcal{T}'_{\mathfrak{n}}, \mathcal{L}'_{\mathfrak{n}})]}$$

By the induction hypothesis, we know that $\vdash \mathit{thys}' \Downarrow \mathcal{E}'$ and that $\mathit{thys}' = \mathit{thys} \setminus (\mathfrak{n}_{\mathit{del}}, \mathit{theory}_{\mathit{del}})$. Thus, $\mathfrak{n} \notin \mathit{thynames}(\mathit{thys}')$, since it is not in *thys*. A combination of this fact and the precondition means that $\mathit{imports}(\mathit{theory}) \subseteq \mathit{thys}'$. Finally, to prove that $\mathit{thys}' \vdash \mathit{theory} \Downarrow \langle \mathcal{T}'_{\mathfrak{n}}, \mathcal{L}'_{\mathfrak{n}} \rangle$ is valid we need to use the closure under minimal environments theorem for evaluation of a theory in a proof document (Theorem 56 on page 222). □

10.4 SUMMARY AND RELATED WORK

10.4.1 *Related work*

In this section, I briefly compare the approach to proof refactoring with some similar programming language approaches:

HASKELL REFACTORING: Li and Thompson (2005) demonstrate a technique for specifying and proving correct the refactorings implemented in the Haskell refactoring tool, HaRe. The authors specify a *generalise definition* refactoring, similar to *generalise tactic*, using a version of the lambda calculus to abstractly represent a Haskell program. The specification is given as a simple transformation on the lambda term version of a program. Their correctness proof shows that the initial and refactored term will reduce to the same value. The authors further extend the lambda calculus to include a module system and use it to specify a refactoring that moves an item from one module to another, a refactoring that I did not specify.

While this approach makes for precise specifications and a precise notion of correctness, it is distanced from Haskell as the language and semantics the specification uses is not the language being refactored, unlike with Hiscript. The abstract nature of the specification language makes for difficult to read specifications, particularly for the module level refactoring.

OPDYKE AND ROBERTS: It is useful to compare Hiscript refactoring with some of the original work on programming language refactoring. Opdyke (1992) was the first approach on refactoring, while Roberts (1999) introduced the first tool, for Smalltalk, the Refactoring Browser. Both approaches specified preconditions in a similar way to my approach, but left the formalities of transformation rules unspecified. Roberts (1999) introduced postconditions as a technique for composing refactorings statically. Postconditions could directly demonstrate that the preconditions of the refactoring to be composed are satisfied, rather than an explicit proof as required for Hiscript.

ROOL REFACTORING: Cornélio et al. (2002) specify refactorings for a language implementing a subset of Java, called ROOL. The approach is based on a set of algebraic laws that define equivalences between programs. The refactoring specifications are then derived laws. This approach gives similar specifications to Hiscript refactorings, though they are not given in the inductive style as a match is assumed. Furthermore, the proofs are algebraic and rely on the set of laws of ROOL, whereas Hiscript correctness proofs are always an induction over the transformation.

GRAPH TRANSFORMATION: A different approach to specifying refactorings is given in Mens et al. (2005), where refactorings are specified as graph rewrites on a graph representation of the program. This approach is similar to POLAR, described in the next chapter and I do not compare it with the Hiscript refactorings.

10.4.2 *Summary*

The past two chapters have presented over twenty proof refactorings and over ten theory refactorings. The refactorings range from simple copying operations, that have a straightforward behaviour and correctness proof, to sophisticated refactorings like conversion of a backwards style proof to a forwards style proof. Furthermore, many of the refactorings build upon each other to create more complex refactorings. A good

example of refactoring composition is the *local to global* refactoring. This refactoring extracts a local lemma or tactic definition from inside a declarative proof to the level of a theory. The refactoring itself is defined to only extract the *top* statement in a proof block, which is a simple transformation that is easy to prove. The more general behaviour of extracting a local lemma or tactic from a proof block can be achieved by combining other refactorings like *generalise tactic* and *swap statements* to move the desired definition to the top of the proof block.

Each of the refactorings were shown to be correct: that they preserve the semantics of the theory in a precise way. This task was relatively trivial for proof refactorings and even for single theory refactorings, but became gradually more complex as the layers of the proof language framework were built upon.

As well as the contribution of over thirty formally specified refactorings, the work in formalising these refactoring specifications led to a number of approaches being developed. In particular, a common pattern of specification and proof was the *modifier* pattern, where the actual work of the refactoring only occurred at one place in the proof or theory. In this case, I could use a standard set of transformation rules to parse through the theory to find the item to refactor. Similar to this pattern, but requiring more work in the proofs was a *modify after* style of refactoring where all proof steps or theory items after a particular step required refactoring. It is in this type of refactoring, and particularly for refactorings that operated on theories, that it was often important to strengthen the correctness property for refactorings. The reason for this was to give enough information in the induction hypothesis to ensure that the refactored theory would evaluate successfully.

While the refactoring specifications are relatively concise and have correctness guarantees, there are a few issues with this approach:

- I use an abstract notion for referencing the items to refactor. In reality, many tools use line numbers or references to the abstract syntax tree to make these precise references.
- These abstract refactoring specifications are specific to the Hiscript language and may not have a direct translation for another proof language.
- To fully specify a refactoring in this way, is very verbose and, for a set of refactorings, contains much repeated steps.
- The implementation of such a refactoring specification may not look exactly like the specification itself.

In the next chapter, I describe a different approach to proof refactoring that promises to solve many of these problems and, moreover, has been implemented in a prototype tool.

11

A FRAMEWORK FOR PROOF REFACTORING

11.1 INTRODUCTION

In the previous chapters, I demonstrated that refactorings could be formalised and proved correct for a proof (document) language, Hiscript. While these refactorings could be implemented the preconditions and transformation rules may not always have a direct implementation and, furthermore, may require a deal of programming, resulting in a gap between correctness of specification and correctness of implementation. This is a difficulty of most refactoring tools and is well-known (Schaefer and de Moor, 2010).

In this chapter, I describe an alternative approach to proof refactoring, which has been implemented in a prototype refactoring tool called Polar (PrOof LAnguage Refactoring). The work described in this chapter was performed equally with Dr Dominik Dietrich, from DFKI Bremen[1].

In designing a refactoring framework, we have four key requirements:

1. Since the theorem proving community is relatively disparate, with many different systems that each have a reasonably small userbase, we wished it to be as widely applicable as possible.
2. Furthermore, given the breadth of proof languages we can't implement all the refactorings that may be required by proof engineers. Therefore, we wish it to be extensible: allowing proof engineers to implement their own refactorings.
3. We want to provide guarantees that the tool will not cause unexpected *semantic* changes to the proof development.
4. Finally, refactorings should be specified in a natural way, so simple refactorings should require only a few lines to implement.

Based on recent research in programming language refactoring and the observation that many of the transformation rules for Hiscript simply traverse through the abstract syntax to find the appropriate syntax to refactor, we based our approach on *graph rewriting*, where declarative rules can directly match the location to refactor. We introduce a graph meta-model into which proofs from different languages can be mapped. We then allow the specification of *abstraction* rules to create an abstract graph that includes only details relevant to a particular refactoring. This abstracted graph can be enriched with semantic information, such as dependencies and it is

1 As a result, I will use 'we' to describe Dominik's and my approach.

to this annotated, abstract graph the refactorings, specified as rewrite rules, can be applied. Then, an experimental back transformation mechanism provides a means to propagate changes back from the abstract representation to the concrete graph and finally back to the syntax.

Our meta-model is expressive enough to allow many different proof languages to be mapped to it, thus making our approach generic. Furthermore, the combination of abstraction and the use of graph rewrite rules makes our refactoring specifications compact and declarative. Finally, the well-understood subject of graph rewriting in combination with a formal semantics for a proof language can allow us to argue about the correctness of refactorings.

CHAPTER MAP The rest of this chapter is structured as follows:

1. In the next section, we motivate and describe the overall approach taken in constructing this refactoring framework.
2. Section 11.3 describes the graph meta-model in detail.
3. Section 11.4 describes the translation to and from the graph model, including our novel bidirectional rewriting approach.
4. Section 11.5 describes the dependency analysis techniques.
5. Section 11.6 gives four example refactorings.
6. Finally, Section 11.7 compares our approach to related work.

CONTRIBUTIONS This chapter has the following original contributions:

1. The design and implementation of a prototype framework for refactoring proof developments. We believe this to be the first dedicated tool for proof refactoring. Our tool, POLAR, currently supports two proof languages and over ten refactorings. POLAR is available at `http://homepages.inf.ed.ac.uk/s0569509/refactoring.html`.
2. Furthermore, POLAR is extensible in two directions: new proof languages and new refactorings can be added.
3. We believe our framework is the only approach in the refactoring community to combine abstraction of irrelevant details with a bidirectional transformation mechanism for obtaining a refactored source document.

11.2 APPROACH

Our approach is best described by the workflow in Figure 11.1, which consists of the following steps:

1. Given an unparsed theory $\mathcal{D}$, a parser function p computes the abstract syntax tree (AST), $\mathcal{A}$, of the theory, which is an ordered tree.

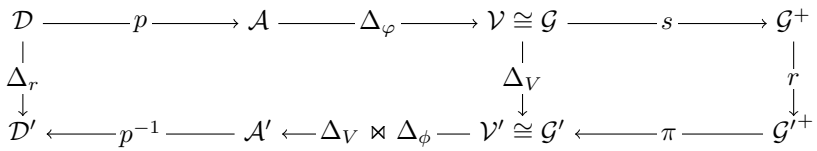

Figure 11.1: Overall workflow of our approach

2. A user-defined abstraction function φ computes the *view* $\mathcal{V}$ of the theory, which is a more abstract representation of the AST represented by an ordered tree. The view contains only the information that is relevant for the refactoring. We denote the difference between $\mathcal{A}$ and $\mathcal{V}$ by Δ_φ. We represent the difference as an edit script.

3. The view $\mathcal{V}$ is translated to an isomorphic unordered attributed graph representation $\mathcal{G}$ that is used by the graph rewriting tool by making the ordering relations among children explicit as shown in Figure 11.2.

4. Using a proof language-specific function s, $\mathcal{G}$ is enriched by semantic information, such as dependencies, resulting in a semantic view $\mathcal{G}^+$. This enrichment of the view is an important part of our approach and allows, for instance, edges to be added between named references, to lemmas, for example, and their definition. These edges can then be followed in a *renaming* refactoring. This enrichment operates solely on the graph representation and does not require further user input.

5. $\mathcal{G}^+$ is refactored, resulting in a modified semantic view $\mathcal{G}'^+$. The refactoring performed is selected by the user and may require additional user-supplied information. For example, a renaming refactoring would require the new name to be supplied.

6. Apply the syntactic projection function π to obtain the modified view $\mathcal{V}'$.

7. The modifications $\Delta_V \bowtie \Delta_\Phi$ between $\mathcal{V}$ and $\mathcal{G}'$ are propagated back to obtain a modified abstract syntax tree $\mathcal{A}'$. The information from Δ_Φ is used to transform the Δ_V so that modifications to the view are transformed to modifications of the AST $\mathcal{A}$.

8. The abstract syntax tree $\mathcal{A}'$ is printed (unparsed) to obtain a modified document $\mathcal{D}'$

The problem of propagating back the modified view (our step 7) to the source is the well-known *view update problem* (Chen and Liao, 2010).

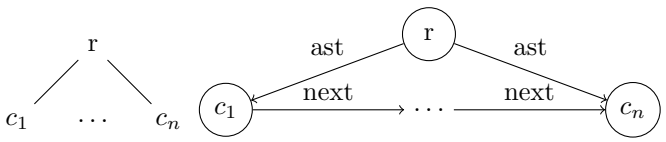

Figure 11.2: Representation of ordered trees as directed graphs

Thus, our approach to refactoring combines two techniques: (i) *graph rewriting* and (ii) a bidirectional *transformation* mechanism. The main advantages of (i) are the use of a formal language to describe refactorings in a language independent format, and the existence of efficient tools. The advantages of (ii) are independence of the actual syntax of the proof language and the support of information hiding, resulting in a lightweight graph representation.

11.2.1 *A running example*

Throughout the rest of this chapter, we use a running example to illustrate each step in the approach. We have implemented two languages in the refactoring framework: Hiscript, as described in Part 1 of this thesis and ΩSCRIPT (Dietrich, 2011). The implementation of Hiscript is restricted to the single theory instance of the language as described in Section 5.2. The running example is a simple theory in both languages. Figure 11.3 is Hiscript and a similar theory in ΩSCRIPT is shown in Figure 11.4.

```
theory set
begin

public tac intro := ⊆—def | ∩—def | id

public lemma comm: A ∩ B ⊆ B ∩ A
proof( intro )
show x ∈ A ∩ B ⊢ x ∈ B ∩ A
  proof( intro )
    show B: x ∈ A ∩ B ⊢ x ∈ B
        by ∩—elim ; ax
    show A: x ∈ A ∩ B ⊢ x ∈ A
        by ∩—elim ; ax
  qed
qed

end
```

Figure 11.3: Hiscript running example

```
theory set

strategy intro := ⊆—def | ∩—def | id

lemma comm: A ∩ B ⊆ B ∩ A
proof( intro )
  assume hyp: x ∈ A ∩ B
  subgoal x ∈ B ∩ A
  proof ( intro )
   subgoal x ∈ B from hyp
    by auto
   subgoal x ∈ A from hyp
    by auto
  qed
qed

end
```

Figure 11.4: ΩSCRIPT running example

The Hiscript theory introduces a single tactic definition called *intro*, which attempts to apply either the definition of subset or intersection; if both fail, the identity tactic is applied, leaving the goal unchanged. The lemma is proved in a backwards fashion, using a familiar declarative-style inside a *proof block*, which operates on a single goal, applying the initial rule before solving the resulting subgoals by the statements inside it. The ΩSCRIPT theory is similar, since both languages are broadly based on Isar, but demonstrates some of differences between the two languages:

- There are some minor differences in syntax: backward steps, for example, are handled in Hiscript using the show command; however, in ΩSCRIPT the command is subgoal.

- Hiscript is a *generic* proof language, which we instantiate with a sequent style notation to describe the proof context. ΩSCRIPT uses a natural deduction style syntax to describe changes of the proof context. Thus, the number of available proof commands differ; ΩSCRIPT, for example, allows assumptions to be named and used directly but this is not possible in Hiscript.
- ΩSCRIPT forbids the combination of forward and backward steps inside a proof block, this is allowed in Hiscript.
- In Hiscript, proof document objects, such as tactics and lemmas, are annotated with a visibility. Only **public** objects are exported. In ΩSCRIPT, all items are exported.
- Intermediate lemmas using the **have** command are introduced into the environment in Hiscript, but introduced to the proof context in ΩSCRIPT. As a consequence, facts can be modified by subsequent tactic applications.

11.2.2 *Example refactorings*

In this chapter, we detail four refactorings:

1. **Backwards to forwards:** translate the lemma *comm* into a forwards style proof, as described in Section 9.12 on page 138.
2. **Rename item:** renaming the tactic *intro* to *tryintro*, as described in Section 8.2 on page 111.
3. **Generalise tactic:** introduce a more general *try* tactic by generalising *tryintro*, as described in Section 8.8 on page 123.
4. **Rename assumption:** a refactoring, only possible in ΩSCRIPT, that renames the assumption *hyp* to *xinAB*.

Each refactoring has previously been described in this thesis so we do not provide detailed motivation or description of behaviour[2]. The resulting Hiscript and ΩSCRIPT theories are shown in Figures 11.5 and 11.6. There are now two tactic definitions: the tactic introduced by generalisation, *try*, and the renamed *tryintro*, which, as a result of the generalisation, is now defined in terms of *try*. The lemma has had each instance of *intro* renamed and the proof block solving $x \in A \cap B \vdash x \in B \cap A$ has been transformed to a forwards style proof. Over the course of this chapter, we will see how these refactorings are performed.

11.3 GRAPH META-MODEL

The graph meta-model provides a source-language independent format, such that different languages can easily be connected to the refactoring framework. Formally, we use attributed, typed graphs with inheritance which enrich standard graphs by the concept of inheritance known from object-oriented programming and attributes

2 Of course, *rename assumption* is not a valid refactoring for Hiscript, but its behaviour is identical to the other renaming refactorings that I have described.

```
theory set
begin

private tac try(X) := X | id
public tac tryintro := try(⊆—def | ∩—def)

public lemma comm: A ∩ B ⊆ B ∩ A
proof( tryintro )
show x ∈ A ∩ B ⊢ x ∈ B ∩ A
  proof
    have B: x ∈ A ∩ B ⊢ x ∈ B
        by ∩—elim ; ax
    have A: x ∈ A ∩ B ⊢ x ∈ A
        by ∩—elim ; ax
    from B A show thesis by tryintro
  qed
qed

end
```

Figure 11.5: Refactored Hiscript theory

```
theory set

strategy try(X) := X | id
strategy tryintro := try(⊆—def | ∩—def)

lemma comm: A ∩ B ⊆ B ∩ A
proof( tryintro )
  assume xinAB: x ∈ A ∩ B
  subgoal x ∈ B ∩ A
  proof
   have l1: x ∈ B from xinAB
    by auto
   have l2: x ∈ A from xinAB
    by auto
   subgoal thesis from l1, l2 by tryintro
  qed
qed

end
```

Figure 11.6: Refactored ΩSCRIPT theory

that can be attached to nodes and edges to store primitive types such as integers or strings. The inheritance on node and edge types allows us to define classes of nodes to simplify analysis and rewriting. For example, any theory items (lemmas, tactics, axioms, definitions) can have a node type that is a subtype of the more general *thyitem* node type. Thus, writing a rewrite rule to match theory items does not need a case for each particular item.

11.3.1 *An example graph*

Before describing the formalities of our graph, we provide an example instance for the Hiscript theory in Figure 11.3.

A particular view of the graph obtained from the example theory is given in Figure 11.7. It is clear that the constructed graph is similar to an abstract syntax tree for the theory; however, there are some notable differences. Firstly, our graph representation allows *attributed nodes*. Thus, we store the names of objects in the theory as attributes in their corresponding node. Additionally, our approach supports hierarchical transformation. In this case, we *abstract* away individual formula representations. The motivation behind this is to only present required details for a refactoring. In the underlying graph model, we have typed graphs. In this graph we see the node types for Lemmas's, Tacdef's and Theory's. What is not visible is the inheritance structure of types. We have a type ThyItem, of which Lemma, Tactic, Definition , Axiom, etc are subtypes. We write Lemma < TheoryItem to represent this relationship.

Figure 11.8 represents the proof block solving the goal x ∈ A ∩B ⊢ x ∈ B ∩A. We again see the same explicitly ordered tree representation for the steps in the proof

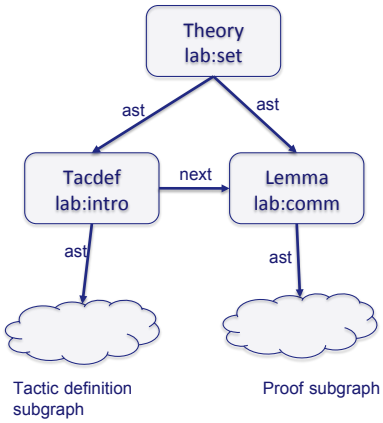

Figure 11.7: The proof graph of Figure 11.3

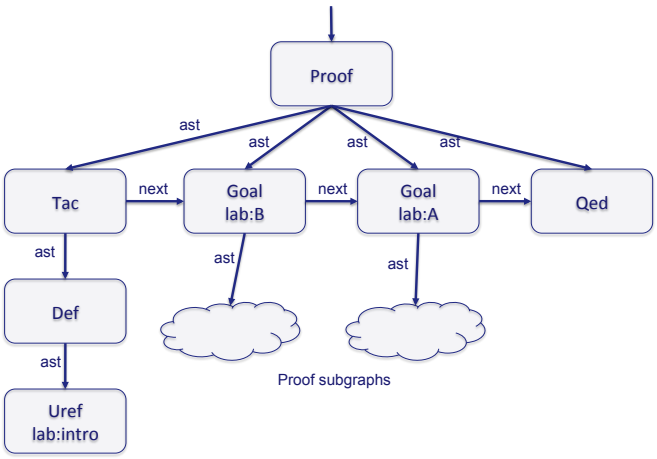

Figure 11.8: The graph representation of a proof block

block; also implicit in the graph is the sub-typing hierarchy with Goal and Qed node types being a subtype of ProofStep. The proof block subgraph introduces two additional elements of the graph structure. Firstly, the proof block introduction tactic (in this case *intro*) is represented as a subgraph. The Def node type represents defined tactics in the language. In order to represent *named references* — to assumptions, tactics, other lemmas etc. — we use the node type Uref, with a label attribute (shortened to lab in the figures). Figure 11.9 shows the equivalent proof block for the ΩSCRIPT example (from Figure 11.4).

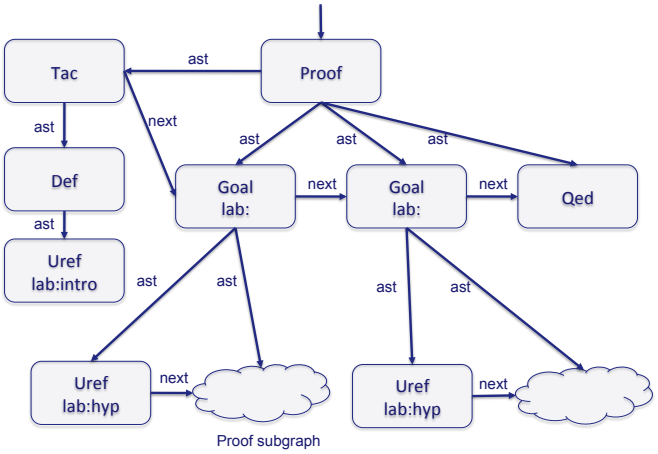

Figure 11.9: The graph representation of a proof block in ΩSCRIPT

The graph representation of the two proof blocks is almost identical. The syntactic differences — **subgoal** and **show**, for example — have been merged in the translation to Goal nodes. Furthermore, the **from** statements in the ΩSCRIPT proof have been mapped to Uref nodes. Note that the attributes storing the **show** statement names are blank in the ΩSCRIPT version, since the **subgoal** statements have no names.

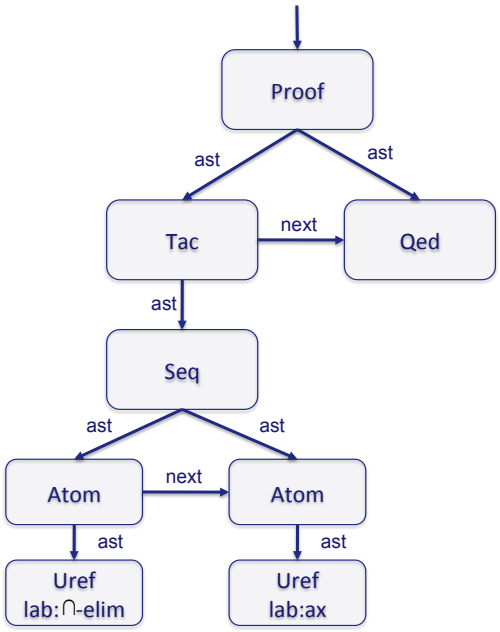

Figure 11.10: **by** statement graph

As a final illustration of our graph representation, consider Figure 11.10. This is the complete graph of the expression by ∩−elim ; ax. Instead of an explicit node type for by statements, we use a degenerate version of a proof block: one without any statements in it.

11.3.2 *Graph model*

The allowable structure of a graph is captured in the form of an *attributed type graph*. The type graph restricts the node and edge types that can occur and link together in the meta-model, and describes the attributes for each node together with their types. Thus, it describes the structure of all its instances in an abstract way and allows us to study relations between different languages. Given a proof language $\mathcal{L}$ and a type graph t, we call an abstraction function φ *admissible wrt.* t and $\mathcal{L}$ iff for all ASTs $l \in \mathcal{L}$ the abstraction $\varphi(l)$ satisfies the requirements imposed by the type graph (formally, the existence of a total graph morphism into a type graph).

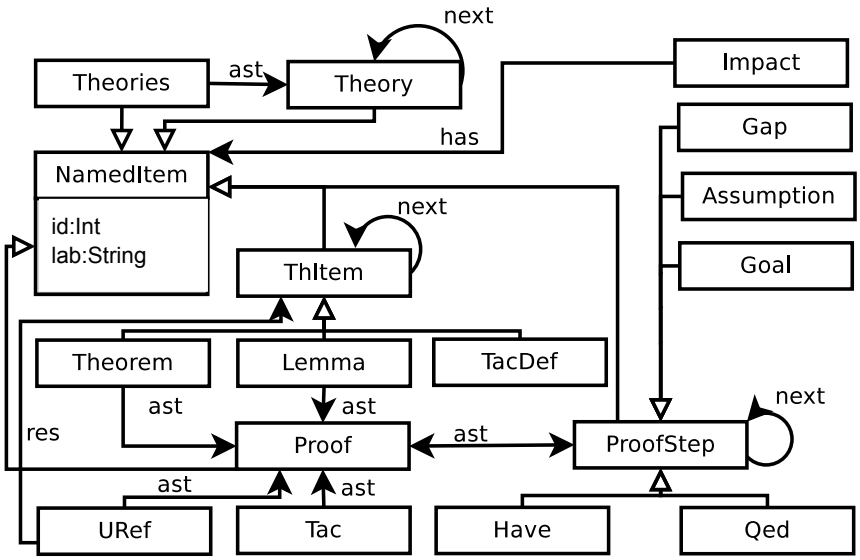

Figure 11.11: Type Graph

Figure 11.11 shows an excerpt of our type graph in a compact notation inspired by de Lara et al. (2007). In the figure, $a \twoheadrightarrow b$ indicates that a is a subtype of b and inherits all the attributes of b, whereas $a \xrightarrow{\texttt{type}} b$ indicates that edges of type `type` are allowed between nodes of type a to type b. This graph shows the main type graph structure for a theory and its containing items as well as the type graph structure for proofs of lemmas. We do not show the section of the type graph that corresponds to tactic expressions, but it is similar.

The graph model is based on an abstract node type *NamedItem* which has two attributes: a label that represents its name and an identifier that is used internally to uniquely identify a node. It then introduces nodes according to the structure of a typical proof document: a node for theories, tactics, proofs and proof steps. Additionally,

a node of type *Impact* is introduced, which is used to attach additional information to the nodes, e.g., failures that are detected by the dependency analysis.

Note that there several possible graph representations which differ on how declarative they are, their size and the involved rewrite rules. For example, the name of a lemma, or any other labelled item, can either be expressed as a label node that has an attribute to store its actual value, but also as an attribute in the node representing the lemma (i.e. Figure 11.12), because each lemma can have at most one label. While both representations contain the same amount of information, they differ in readability and the number of nodes that need to be traversed during the analysis. They also have an impact on the rewrite specifications that we will see later: nodes can be matched declaratively, while attributes cannot be matched in preconditions of rules.

Figure 11.12: Representations of a proof step

As another example, attributes of proof steps (such as **from**) can either be expressed as fields in the corresponding node, or be represented as separate node. The latter solution allows for a declarative matching, but also introduces more cases during the analysis. Our particular graph representation is driven by the following design principles:

1. Branching nodes are avoided as they complicate the analysis (e.g. definition lookup); therefore, multiple assumptions of an assume step are linearly ordered in the graph model.
2. The possibility to represent faulty proof scripts (e.g. with duplicate labels).
3. Each proof command has a corresponding node type in the meta-model.
4. Conciseness is preferred over declarativity.

Note that **qed** is represented explicitly as the node type Qed — whereas the equivalent theory level construct **end** is not — because in some proof languages, tactic expressions can be attached to it. For example, in ΩSCRIPT one can write **qed by** auto as a valid command. The semantics enforce that the supplied tactic must solve all remaining goals in the proof block. Hiscript, for example, does not support this construct. We finally point out that we do not consider the current meta-model to be a 'final' representation. We expect that the process of writing more refactorings and connecting additional languages will induce changes.

11.4 ABSTRACTION AND BACK TRANSLATION

Since we allow different mappings to the meta-model for each language, we provide a generic abstraction mechanism to perform simple manipulations on the original AST, such as hiding specific subtrees. This allows us to experiment with different graph representations for different refactorings — in particular, to work with small and

human-readable graphs — but it also requires a more sophisticated change model that propagates back the changes made by a refactoring on the abstract graph representation to the original AST. We first describe the process by which we transform an AST into the view and thus to the graph, then describe the back translation process.

11.4.1 *Obtaining the view*

ASTs are transformed to their view by the application of abstraction rules, which operate on the AST of a well-formed proof document and result in an attributed tree. Abstraction functions are specified by a list of rewrite rules $\varphi := (s \to s)^+$ where each side of the rule has the following form:

$$s := v(:t)? \mid (n\ r_1\ \ldots\ r_n) \mid (@\ s\ v) \mid r^* \mid r^+ \tag{11.1}$$

Here, $v : t$ denotes a variable of type t, n the type of a non-atomic node of the AST with children r_i; $?$, $+$, and $*$ are the standard regular expressions to express repetitions. @ is a special symbol that allows the user to introduce new attributes to the attributed tree: a rule $v \to (@\ a\ v)$ introduces a new attribute with name a and value v provided that v is a leaf node (as attributes can only hold atomic values). We require that no variable occurs twice in either l or r, that function symbols in l are a subset of the constructor symbols in the AST, and that function symbols in r correspond to the signature of the meta-model. The first condition ensures that no content is duplicated, making the back translation difficult, while the other two conditions express syntactic constraints on the rules.

The view is then obtained by traversing $\mathcal{A}$ top down and trying to apply the rewrites in a specified order:

$$\mathcal{V} := \mathsf{topdown}\ (\mathsf{try}\ \varphi_1\ \ldots\ \varphi_n) \tag{11.2}$$

To illustrate, we describe the set of rules for Hiscript and their application on the AST generated by the example theory in Figure 11.3. First, Figure 11.13 shows the raw AST for the theory. We have elided the subtrees containing the sub-ASTs of the formula, representing those subtrees by ellipses. As is usual with ASTs, the node types are that of the lexer tokens, and it is these values that are matched by the set of rewrite rules. The bracketed numbers are unique, persistent identifiers for the tokens generated by the parser, which we use to identify nodes.

An example abstraction rule for obtaining the view is:

```
(TAC visib label tac arg?) -> (TACDEF (AT "lab" label) tac arg?)
```

We read this rule as matching a tree rooted with the lexer type TAC and at least three subtrees: one for the visibility, one for the name of the tactic, and one for the tactic definition itself. There is also an optional subtree for any parameters for the tactic, matched with the optional ? attribute. To illustrate the abstraction process, Figure 11.14 shows a small portion of the full AST corresponding to the tactic definition and Figure 11.15 shows the view resulting from applying the rule above. This abstraction rule performs three changes:

1. It performs a renaming of the lexer type TAC to TACDEF, which is the type of the equivalent node in the graph representation.

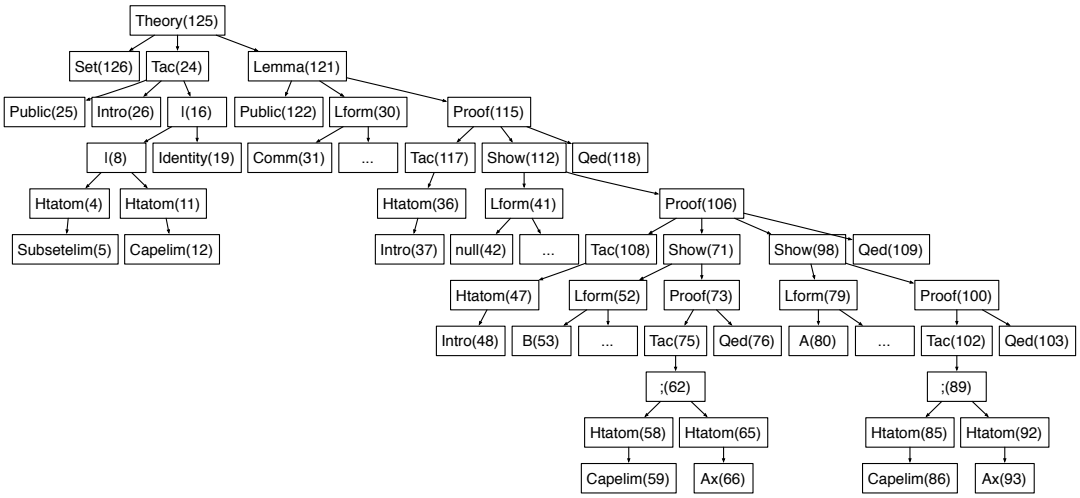

Figure 11.13: AST of the Hiscript proof with formula AST not shown

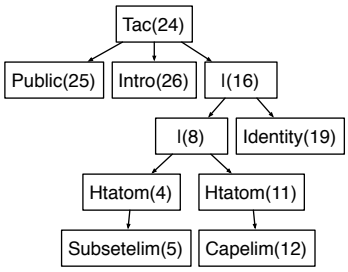

Figure 11.14: AST of a tactic definition

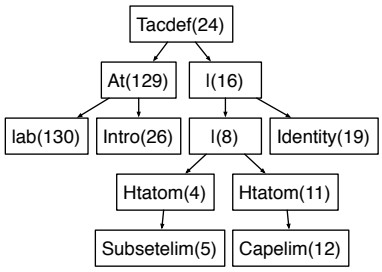

Figure 11.15: AST after applying rule

```
abstraction rules :
(LFORM label term) -> (AT "lab" label)
(THEORY label items*) -> (THEORY (AT "lab" label) items*)
(TAC visib label tac arg?) -> (TACDEF (AT "lab" label) tac arg?)
(HTAC visib label tac arg? ) -> (TACDEF (AT "lab" label) tac arg? )
(LEMMA visib lform proof) -> (LEMMA lform proof)
(SHOW lform proof) -> (GOAL lform proof)
(FROM (FORMULA f) tac argss) -> (FROM tac argss)
(SEMICOLON t1 t2) -> (SEQ t1 t2)
(TENSOR t1 t2) -> (TENS t1 t2)
(BAR t1 t2) -> (ALT t1 t2)
(ARGS arg*) -> (TACPAR (AT "lab" arg))*
(HTATOM label) -> (HTATOM (UREF (AT "lab" label)))
(HTLEM label) -> (LEM (UREF (AT "lab" label)))
(HTDEF label args) -> (DEF (UREF (AT "lab" label) ) args)
(HTDEF label) -> (DEF (UREF (AT "lab" label) ))
(HTVAR var) -> (HTVAR (UREF (AT "lab" var)))
(HTLAB lab tacexpr) -> (LAB (UREF (AT "lab" lab) ) tacexpr)

default rules :
(SHOW (PROOF proof)) -> (SHOW (FORMULA THESIS) (PROOF proof))

propagation rules :
(AT "lab" label ) -> label
```

Listing 11.1: Abstraction rules for Hiscript

2. It deletes the visibility subtree from the AST, as it is not needed for the graph representation (for the refactorings we perform).

3. Finally, it introduces an *attribute* to store the name of the tactic being defined. It is represented as a tree with the root type *AT* — the AST representation of @ — and the attribute name and value as children.

Note that the *view* is an ordered tree. After the abstraction rules are applied, it is transformed into the isomorphic unordered, attributed graph representation. In particular, we transform the trees rooted using the *AT* type to attributes of its parent. Furthermore, the node identifiers have persisted (even through renaming) and where new nodes have been introduced, a new, unique identifier has been added.

11.4.2 *Constructing the Hiscript view*

The full set of rules for Hiscript is shown in Listing 11.1. For our running example, Figure 11.13 shows the original AST $\mathcal{A}$ of Hiscript; the view $\mathcal{V}$ that is obtained by applying the abstraction rules to $\mathcal{A}$ is shown in Figure 11.16. The abstraction specification consists of three parts: a rule part, where the actual abstraction rules are specified, a default part, which specifies default values that are used to fill in holes in the back propagation process in case abstracted values cannot be reconstructed from the original theory, and a propagation part which is used to correctly propagate modifications

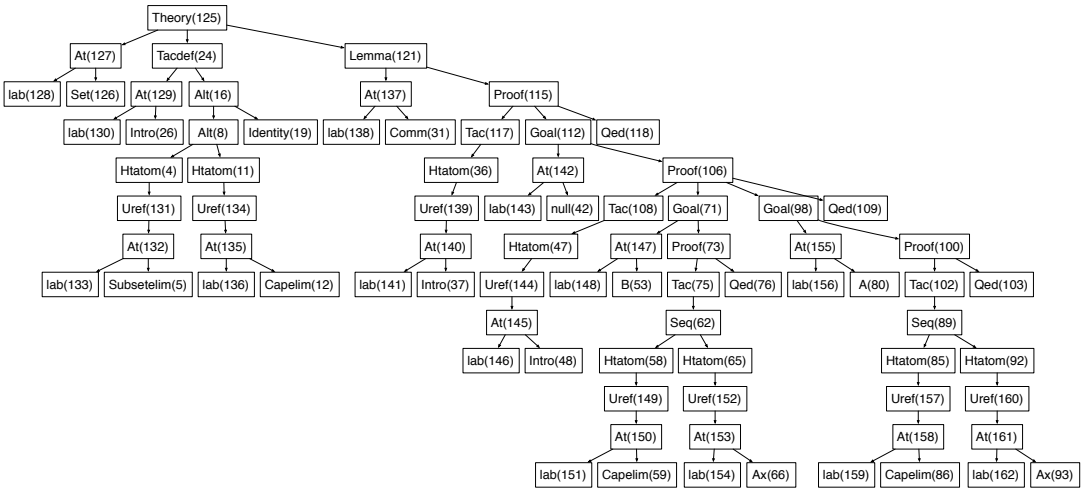

Figure 11.16: The view obtained from abstraction

inside the original AST. Let us point out that our transformation is a true abstraction: in this case, we abstract away individual formula representations and visibility of theory items. The motivation behind abstraction is to only present details of a theory that are required for a particular refactoring. This abstraction process is the reason behind our sophisticated back propagation theory described in the next section. In the example, we see a typical abstraction process that is used for structural refactorings: the abstraction rules abstract away the formulas, introduce attribute nodes for labels, and perform the renamings tac $\mapsto$ tacdef, hitac $\mapsto$ tacdef, and show $\mapsto$ goal to fit into the meta-model.

11.4.3 *Back propagation: from the view to concrete*

Assuming we have already refactored our abstracted graph, the changes in the abstract representation must be propagated back to the AST and finally back to the theory. The key problem for propagating the modified view back to its source is that the abstraction is not one-to-one, meaning that some information is lost: the formulae, for example.

In general, given a proof node to be converted (from abstract to concrete), there are two possibilities: (i) the proof node existed already in the original graph, in this case, the abstracted information can be reconstructed from the original graph; and, (ii) the proof node was added by the refactoring operation. In this case, we ensure that, if necessary, a default value is provided to keep the document well-formed.

Our solution is based on the computation of differences in the form of an *edit script*:

Definition 13 (edit operation, edit script)**.** An *edit script* is a sequence of the following basic edit operations that convert one tree into another

1. $\mathsf{delete}(m)$ deletes the tree rooted in node m, where m is not the root node.
2. $\mathsf{insert}(n, k, m)$ will insert the tree rooted in node m to be the kth child of the node n.

3. insert-after(n, k, m) will insert the tree rooted in node m to be the right sibling of k with parent n.

4. move-before(n, k, m) moves the tree rooted in node m to be the left sibling of k with parent n.

5. move-after(n, k, m) moves the tree rooted in node m to be the right sibling of k with parent n.

6. update(m, n, v), which changes the attribute n of node m to v.

In our approach, two edit scripts are generated: one between the concrete AST and the view (written Δ_φ) — obtained by the abstraction process — and the other between the view and the modified view (written Δ_V) — obtained by the refactoring process.

As a simple example, the edit script generated by the abstraction rule for tactic definitions is shown:

```
delete(25)
insert(24,0,129)
update(24,content,Tacdef)
moveAfter(129,130,26)
```

This says:

1. First delete node 25: the **public** node.

2. Then insert node 129 as the zeroth child of node 24.

3. Then update the 'content' attribute of node 24 to read 'Tacdef'. The content attribute stores the type of the node.

4. Finally move the tree rooted at 26 to be the right sibling of node 130, with parent 129. This moves the name to the value position of the attribute.

This edit script transforms the AST in Figure 11.14 to the view in Figure 11.15. The refactoring process constructs its own edit script. The complexity of the back propagation process lies in the fact that the refactoring process induces *changes* in the edit script Δ_φ. This is because a refactoring can insert, move or delete subtrees at will so, for example, what was previously the kth subtree is now the $k+1$th subtree.

To compute the differences efficiently, we use persistent identifiers for nodes. These identifiers are used to track the origins of the nodes, i.e. the changes of the document. In particular, renamings that are difficult to detect with no identifiers can easily be detected. Within our implementation, the identifiers correspond to the internal identifiers that are constructed during the parsing of the document and are never touched by the user.

To propagate back the modified view to the source level, we proceed by the following steps: (i) deletes and updates on the view are applied to the source; for updates, renamings are applied. (ii) Moves of the view are translated to moves of the source; child positions are adapted based on the diff computed by the abstraction function. (iii) Inserts on the view are propagated to inserts on the source, child positions are

adapted as well. To compute default values, the default rules are applied to the new elements. (iv) Finally, the propagation rules are applied in order to clean the modified AST.

Our back-propagation approach is experimental and we plan to further develop the theory and practice behind the approach, but has been sufficient for the refactorings that we have implemented for both the Hiscript and ΩSCRIPT languages. In particular, we wish to compare our approach with the approaches used in the field of bidirectional transformations, for example, Stevens (2008); Hofmann et al. (2012); Bancilhon and Spyratos (1981).

11.5 DEPENDENCY ANALYSIS

At this point in the POLAR framework, we have parsed and abstracted a theory into its *view* and translated that view to the isomorphic graph representation that was described in Section 11.3. The next step is to enrich the graph with semantic dependencies, then apply the refactoring. Both these tasks are performed by graph rewriting. This section describes the dependency analysis and the next describes the refactorings themselves

11.5.1 *Types of dependencies*

Within a formal theory, there are many implicit dependencies between the statements that need to be respected when applying a behaviour-preserving transformation. For example, changing a name of a variable at some place might require to change it at another place as well.

Dependency analysis has the goal to make dependencies due to interconnections between statements explicit. Usually, these dependencies are statically identified using control flow and data flow analysis, which can be performed based on a program dependency graph (see Ferrante et al. (1987)). A systematic review of existing solutions can be found in Arias et al. (2011).

We follow this common approach of static analysis, and enrich the syntactic graph by semantic edges, resulting in an abstract semantic graph. These edges are used in the preconditions of the rewrite rule to check whether a refactoring can be applied or not, and to propagate changes conveniently.

As a concrete example in theories, a tactic reference is connected to its associated definition, a theorem to its use, etc. We distinguish two kinds of dependencies: *explicit dependencies* and *implicit dependencies*. Explicit dependencies are dependencies that hold between two objects and can thus be represented in the graph by an edge. Implicit dependencies are dependencies that hold between several other items, such as the dependency that each label must be unique inside its context. Such dependencies are not explicitly introduced into the graph but are realised by graph patterns inside our refactoring specifications (described in the next section). For example:

- To rename a tactic, we need to know precisely all the *call-sites* to rename it there. This is a precise dependency of a label reference to definition.
- To move lemma B above lemma A, we need to know that it doesn't use the parent lemma. This is a dependency between two proof document items. In

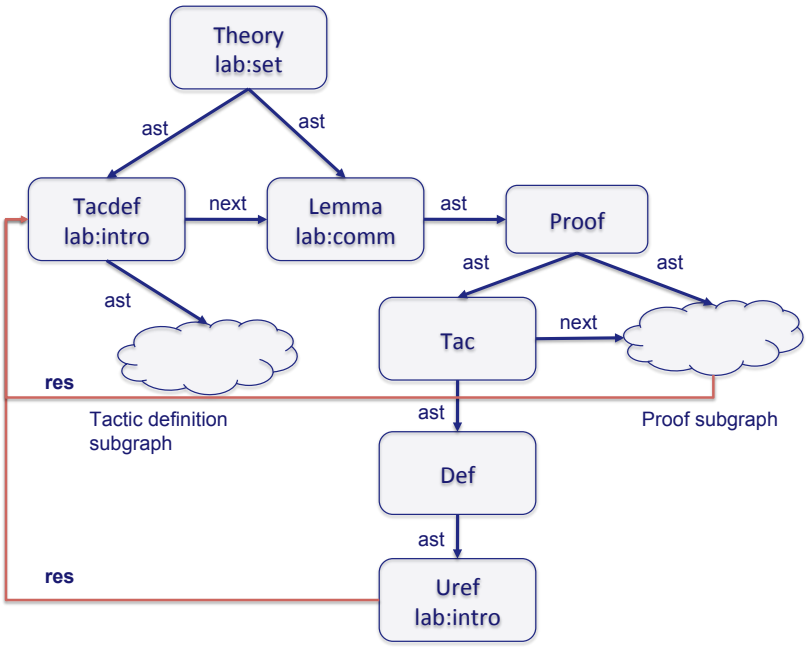

Figure 11.17: Partial enriched graph

fact, we could see it as a higher-level dependency saying that nowhere in the proof of lemma B is there a reference to lemma A.

Dependencies can also be classified according to their scope: a dependency may only be valid at the term level, at a proof block level, or at the proof level; however, it may be valid throughout the theory or even throughout the whole proof document. In our example Hiscript theory in Figure 11.3, we see theory-wide dependencies from the two applications of *intro*. The rest of the tactics in the theory are *atomic tactics* in the Hitac language. The ΩSCRIPT example in Figure 11.4 contains further dependencies between the assumption hyp: $x \in A \cap B$. and its two uses.

11.5.2 *Dependency analysis in* POLAR

In POLAR, we enrich the graph by performing graph rewrites to to add edges of type res (for resolve) *from* the reference *to* the definition.

To illustrate the result, Figure 11.17 shows a part of the enriched graph from our running example. The graph shows the top level of theory and part of the first proof block of the lemma, including the *intro* tactic. The dependency analysis has added an edge of type res linking the Uref node to the Tacdef node. Furthermore, the analysis adds a second reference from the nested proof block that is not shown.

We represent the dependency analysis as rewrite specifications. The following is the top-level analysis rule for Hiscript:

```
rule PLAnalysisHiscript {
 modify {
   exec (PLResolveReferencesTac* );
   exec ( PLResolveReferencesFrom* );
   exec ( PLResolveReferencesTacVar* );
```

```
  }
}
```

The syntax we use is that of GrGen, the graph rewriting tool that Polar is built on top of (Geiß et al., 2006). Rules in GrGen typically have two sections: a **pattern** to match, which forms the precondition of the rewrite rule and binds variables to graph elements; and a **modify** part that performs the rewriting. The rule PLAnalysisHiscript has no preconditions, so we omit the pattern in this case. The modifications are then performed sequentially and the $*$ operator means apply the rule as often as the preconditions match. Thus, this rewrite rule will:

1. Apply the PLResolveReferencesTac rule as often as possible. This rewrite rule looks for references to theory items, such as lemmas and tactic definitions and then looks back through the graph recursively to find the definition. Its behaviour is general in three ways:
   a) It operates on the body of tactic definitions and in proofs.
   b) It resolves references to both tactics and lemmas.
   c) It resolves references to locally defined tactics and lemmas.

   We give full details of this rule below.
2. Then, apply the PLResolveReferencesFrom rule as often as possible, which resolves dependencies introduced by **from** statements.
3. Finally, the PLResolveReferencesTacVar rule analyses the local dependencies between tactic variables and their parameters. In the definition below, for example, it adds res edges between the formal parameter X and its uses.

   public tac ALL(X) := X $\otimes$ ALL(X) | $\langle\rangle$

The specification for the rule PLResolveReferencesTac is shown in Listing 11.2 and the auxiliary rule ItResolveRefTac that adds the dependency edge is shown in Listing 11.3.
The rule PLResolveReferencesTac contains a pattern that matches a graphlet:
defn : Tacref –:ast–> uref : Uref.
The syntax means a graph fragment of the form shown in Figure 11.18. The expression var : type allows us to bind a variable name to a particular node so we can refer to it later in the rewrite rule. We then have two **negative** conditions that check:

1. That the reference has not already been resolved.
2. That the reference has not already been analysed and found to be *unresolvable*.

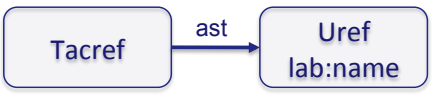

Figure 11.18: The pattern matched by PLResolveReferencesTac

The node type Tacref is a super type of the node types Def, which represents references to defined tactics and Lem, which represents references to lemmas. This is an

```
rule PLResolveReferencesTac{
  defn:Tacref —:ast—> uref : Uref;
  negative {
    uref —:res—> someLabel:NamedItem;
  }
  negative {
    uref —:has—> impact:Impact;
  }
  ipr : ItResolveRefTac(defn, uref);
  modify {
    ipr();
  }
}
```

Listing 11.2: Example analysis rule

example of the type inheritance in our graphs being utilised to make our rules more generic. The final part of this rule:

```
ipr : ItResolveRefTac(defn, uref);
modify {
  ipr();
}
```

calls the auxiliary rewrite pattern, supplying, as parameters, the two nodes that were matched by the pattern. The modify part will trigger the rewriting in the pattern ItResolveRefTac.

The idea of ItResolveRefTac is to traverse the graph until a node defining the reference is found. If it is found, the dependency edge is added; if it is not found, and we hit the top of the theory, then an *impact* node is added to signal an unresolvable reference and to ensure termination. The rewrite pattern defining the behaviour of ItResolveRefTac is split into five *alternatives*. Each alternative consists of a pattern to be matched and a modification should that pattern match. The pattern requires parameters current, which is initially called with the Tacref node, and uref, which is initially called with the node storing the name reference to be resolved.

The pattern is recursive. Starting from the reference, we pass right to left through the next edge relation until there are no more. Then, we move one level up in the AST and repeat the process. This recursive behaviour is captured by the rules PassThroughLevel and PassThroughNextLevel. There are three base cases:

FoundDefExistsThisLevel. This rule has a pattern that is matched if the current node is directly connected by a next edge to a node with a name that is identical to the name of the reference we wish to resolve. If this pattern matches, a res edge is added between the two.

FoundDefExistsNextLevel. This rule is similar only it matches the case when the node one level above the current in the AST has the appropriate name.

```
pattern ItResolveRefTac(current:Node, uref:Uref)
{
  alternative {
    FoundDefExistsThisLevel{
      item:NamedItem -:next-> current;
      if{item.name == uref.name;}
      modify{
        emit("Found the definition and returned it.");
        uref -:res->item;
      }
    }
    FoundDefExistsNextLevel{
      item:NamedItem -:ast-> current;
      if{item.name == uref.name;}
      modify{
        emit("Found the theory item and returned it.");
        uref -:res->item;
      }
    }
    PassThroughLevel{
      pred : NamedItem -:next-> current;
      ipr:ItResolveRefTac(pred,uref);
      modify {
        ipr();
        emit("Passing through the next relation, looking for definition.");
      }
    }
    PassThroughNextLevel{
      negative { pred : NamedItem -:next-> current; }
      pred : NamedItem -:ast-> current;
      ipr:ItResolveRefTac(pred,uref);
      modify {
        ipr();
        emit("Passing up a level, looking for definition.");
      }
    }
    CompleteFail{
      pred : Theory -:ast-> current;
      modify{
        uref -:has-> impact:Impact;
        emit("Unresolvable reference.");
      }
    }
  }
  modify {}
}
```

Listing 11.3: Auxiliary analysis rule

CompleteFail. If the current node is the top-level theory node, it means that we have passed through the theory graph without finding a matching name, thus the reference is unresolvable.

The type of the nodes matched in FoundDefExistsThisLevel, for example, allow this analysis rule to be generic, since it matches

```
pred:NamedItem −:next−> current;
```

where NamedItem is a very general node type and can represent nodes for **have** statements, a **lemma**, or local or global **tac** definitions uniformly.

Since Hiscript has a single namespace — tactics and lemmas can't have the same names — this analysis rule is suitable for references to both tactics and lemmas. We say that analysis rules are proof language specific because this may not always be the case and a separate namespace will require different analysis rules; furthermore, a language with different scoping rules may need its own analysis rules. The genericity in our approach stems from the fact that once these dependencies have been made explicit, the same refactoring should be applicable to different languages.

11.6 REFACTORING

To this enriched graph, we apply the refactorings. To illustrate the approach, we describe the *rename item* refactoring in the next section. Then, in Section 11.6.2 to Section 11.6.4 we briefly describe the specification for *backward to forward*, *generalise tactic*, and *rename assumption*.

11.6.1 *Rename an item*

The refactoring rewrite rule, again in the GrGen syntax, is shown in Listing 11.4. The rule takes two parameters: the *item* to rename and the new name that has been supplied. The rule itself contains two **negative** conditions that express the precondition for this refactoring: that no object already exists with the supplied name. Then, the rewriting part of the rule first matches every instance of a res edge to a reference and renames the reference using the **iterated** language construct. Finally, the name of the definition is itself changed.

11.6.2 *Backward to forward*

Proofs can be formulated in forward-style by deriving new facts, or in backward-style by introducing new subgoals. Typically, during proof search the backward-style is preferred due to its goal-directedness, while the forward-style is often preferred for presentation. The proof refactoring `Backward2Forward` automates the process of converting a proof block in backward style to an equivalent proof block in forward style.

For simplicity, we assume that the proof block has the form given in Figure 11.19 *left* (this is indeed the structure of the proof blocks in ΩScript, however, Hiscript allows additional forward statements and procedural statements inside the block; in this case, the same refactoring is possible). For these proof blocks, it is possible to

```
rule PLRenameLabel(item:NamedItem, var newname:string)
{
  negative {
    defnode:NamedItem;
    :SearchContextAbove(item, newname, defnode);
  }
  negative {
    defnode:NamedItem;
    :SearchContextBelow(item, newname, defnode);
  }
  iterated {
    item <-:res- uref:Uref;
    modify {
      eval { uref.name = newname; }
    }
  }
  modify {
    eval { item = newname; }
  }
}
```

Listing 11.4: Rename item refactoring rewrite rule

transform them to the form shown in Figure 11.19 *right*, provided that l_i are fresh with respect to the context and are not captured at a lower level. As we see in Listing 11.5 the pattern can concisely be translated to a rewrite rule specification. In this specification, we see the precondition that there cannot be any procedural steps in the backwards proof expressed as:

```
negative{ proof -:ast-> :Apply; }
```

Here, we use the **iterated** construct to match the specified subpattern (a show statement) as often as possible and thus allows for a direct realization of the ellipses notation. The refactoring behaves slightly differently depending on whether the backward step being transformed already has a name. After every backward step has been transformed, a **from** statement is introduced with:

```
modify {
  delete(e);
  proof -:ast-> newproof:From -:ast->tac;
  newproof -:ast-> newargs:Args;
  tac -:next-> newargs;
  exec ( InsertBefore(newproof, qed) );
  exec (DeleteOrder(tac));
}
```

The from statement performs the forward step and its arguments are exactly the have statements created within **iterated**.

```
rule ConvertBackwardProof2ForwardHS(proof:Proof)
{
  proof −:ast−> qed:Qed;
  proof −e:ast−> tac:Tac;
  negative{ proof −:ast−> :Apply; }

  iterated {

    proof −:ast −> goal:Goal;
    alternative {
      HasName{
       if {goal.name != null;}
       modify {
        have:Have;
        newargs −:ast−>uref:Uref;
        eval{uref.name = goal.name;} //Add a reference to the have statement
        eval{have.name = goal.name;} //Name the have statement after the show
        exec ( ReplaceNode(goal, have) );
       }
    }
    NoName{
     if {goal.name==null;}
     modify{
       have:Have;
       exec ( AssignFreshLabelAndRef(have, newargs) );
       exec ( ReplaceNode(goal, have) );
     }
    }
      }
        modify{}
  }
  modify {
    delete(e);
    proof −:ast−> newproof:From −:ast−>tac;
    newproof −:ast−> newargs:Args;
    tac −:next−> newargs;
    exec ( InsertBefore (newproof, qed) );
    exec (DeleteOrder(tac));
  }
}
```

Listing 11.5: Backward to forward refactoring rewrite rule

```
proof (t)
  show g1 p1
  ⋮
  show gn pn
qed
```

```
proof (idtac)
  have l1 : g1 p1
  ⋮
  have ln : gn pn
  show thesis from l1,...,ln
       by t
qed
```

Figure 11.19: Refactoring scheme before and after

```
...
public tac mytac() :=   ...
    tsub  ...
...
```

```
...
public tac mytacgen(X) := ...
    X  ...
public tac mytac() :=
    mytacgen(tsub)
...
```

Figure 11.20: Refactoring scheme before and after

11.6.3 *Generalise a tactic*

As a final example refactoring, we describe a simplified tactic generalisation refactoring. This refactoring will take a tactic definition as shown in Figure 11.20 *left* and transform it to the form of Figure 11.20 *right*.

The refactoring specification shown below assumes that there are no other tactic variables in the tactic to generalise, which is less powerful than the Hiscript refactoring described in Chapter 9. The specification is shown in Listing 11.6.

11.6.4 *Renaming an assumption*

The power of our graph rewriting approach means that assumptions can actually be renamed using exactly the same refactoring as lemma renaming, described in Section 11.6.1.

11.7 SUMMARY AND RELATED WORK

11.7.1 *Related work*

We compare several closely related approaches:

GRAPH REWRITING: In the domain of programming languages, (Mens et al., 2005) was the first to show that graph rewriting provides a suitable framework to express refactorings. Our approach is similar, but focuses on genericity and simplicity of the graph representation, which is achieved by a language-independent graph meta-model and abstraction rules. An important aspect in using a graph

```
rule GeneraliseTactic (td:Tacdef, ti : TacItem,var varname:string , var
    gentacname:string){
    td —:next—> suc:ThItem;
    td —:ast—> roottac:TacItem;
    prevexpr : NamedItem —e:ast—> ti;
    hom(prevexpr,roottac);
    modify{
      //First create the new tactic
      newtac:Tacdef —:ast—> newd:Def —:ast—> uref:Uref;
      eval{ newtac.name=td.name;} //add the name as the old name
      eval{ uref.name = gentacname;} //add the name to the def tac call
      newd —:ast—> targ:Targs;
      uref —:next—> targ;
      targ —:ast—> ti; //The argument to the tactic is the new item

      eval{ td.name=gentacname;} //rename the generalised tactic
      //Now replace the proof with a variable
      prevexpr —:ast—> tv:Htvar —:ast—> uref2:Uref;
      eval{ uref2.name = varname;}
      delete(e);

      //Then add the tactic parameter:
      td —:ast —> tp:Tacpar;
      eval{tp.name=varname;}
      tp —:next—> roottac;

      //Finally , map the next relations among theory items
      td —:next—> newtac —:next—> suc;
    }
}
```

Listing 11.6: Generalise Tactic refactoring rewrite rule

representation is how difficult it is to translate the actual language to the graph representation and back. Due to the restriction of our approach to proof languages, it is possible for us to keep the abstract representation very close to the abstract syntax tree, but the graph representation described by Mens is more abstract. Our approach also differs in dependency calculation. We take the approach of specifying analysis functions as graph rewrite rules to be applied to the graph, but Mens et al. (2005) generate this information in the translation of the program into the graph representation. Finally, as our graph representation is closer to an AST representation of a proof language and our language allows for abstraction of unnecessary information, our specifications are arguably more compact and readable than that of the program graph representation.

GENERIC REFACTORING: Closely related to our work is a domain-specific programming language called JunGL, designed to enable a programmer to write their own refactorings (Verbaere et al., 2006). The language is generic in the same sense as ours: to instantiate it for a different object language, a type-graph and parser simply needs to be provided. Where we use graph rewriting to perform the refactorings, JunGL has a number of built in language constructs for adding, removing and modifying edges, which can be composed using a functional programming language. JunGL uses a logic query language to express predicates and match parts of the program graph. One of the benefits of this approach is that it provides a direct representation of the recursive patterns that we employ in POLAR as path queries. In contrast to existing approaches, we explicitly allow for information hiding by abstraction, based on bidirectional transformation. To our best knowledge, this combination has not yet been explored in the literature.

META-MODELS: There are two important techniques that make refactorings generic or language independent, namely the use of *meta-models* as well as *generic* analysis functions that can be instantiated for different languages. This has the advantage that it becomes easier to adapt the refactoring tool to new languages. An important aspect in meta-modelling is how difficult it is to translate the actual language to the meta-model and back. Due to our restriction to proof languages, it becomes possible to keep the abstract representation very close to the abstract syntax tree. In Bell Canada (2000), abstract semantic graphs (ASGs) are introduced. An abstract semantic graph represents an abstract syntax tree (AST) together with additional semantic information. In Lämmel (2002), generic traversal functions were proposed that allow the definition of analysis functions in a generic way. Moreover, the meta-models FAMIX and MOON have been proposed in Tichelaar et al. (2000); Nozal et al. (2006) as a language-independent representation of object-oriented languages.

11.7.2 *Summary*

This chapter presented POLAR: a concrete framework for refactoring formal proof developments in a *generic*, *extensible*, and *declarative* way based on graph rewriting and bidirectional transformation to and from a graph meta-model.

Our meta-model is expressive enough to allow many different proof languages to be mapped to it, thus making our approach generic. Furthermore, the combination of abstraction and the use of graph rewrite rules makes our refactoring specifications compact and declarative. New refactorings can be written as graph rewrite rules in GrGen, using an intuitive language. Finally, the well-understood subject of graph rewriting in combination with a formal semantics for a proof language can allow us to argue about the correctness of our refactorings.

Part III

CONCLUSIONS

12

CONCLUSIONS AND FUTURE WORK

12.1 INTRODUCTION

In this final chapter, I summarise the work presented in this thesis and sketch a number of directions for future work, for both the proof language framework and the refactoring parts. A summary of the Hiscript framework introduced in Part 1 is given in Appendix B. In the next section, I give a similar summary of the approaches to refactoring described in Part 2. I then present some suggestions for future work for both parts. Finally, I conclude in Section 12.4.

12.2 SUMMARY OF REFACTORING APPROACHES

In Part 2, I introduced three approaches to proof refactoring. In this section, I briefly summarise each approach using an example refactoring: *rename lemma*. A theory like that shown in Listing 12.1 can be refactored to the theory in Listing 12.2 and any uses of *lemma1* will also be renamed elsewhere in the theory and any theories that import *set*.

INFORMAL CATALOGUE: This approach follows a tradition in programming language refactorings where refactorings are informally specified and a *recipe* is provided for performing that refactoring. To illustrate, the following recipe performs a lemma renaming from Section 8.2:

- ◇ Create a new lemma with the new name.
- ◇ Copy the body of the old lemma to the new one.
- ◇ Check your proofs still replay. This ensures that the new name you have chosen is a fresh one.
- ◇ Find all references to the old lemma and replace them with a reference to the new one. Replay the proof after each change.
- ◇ Delete the old lemma.
- ◇ Replay proofs a final time to ensure that all references to the old lemma have been removed.

In this approach, the focus of a refactoring is on *testing* to ensure that each step of the process does not break any proofs in the theory.

FORMAL SPECIFICATIONS: This approach uses the formal syntax and semantics of Hiscript to define a set of transformation rules for each refactoring that, as long

```
theory set
begin
 ...

public hitac intro := ⊆—def | ∩—def | id

public lemma lemma1: A ∩ B ⊆ B ∩ A
proof( intro )
show x ∈ A ∩ B ⊢ x ∈ B ∩ A
    proof( intro )
      show x ∈ A ∩ B ⊢ x ∈ B
        by ∩—elim ; ax
      show x ∈ A ∩ B ⊢ x ∈ A
        by ∩—elim ; ax
    qed
qed

 ...
end
```

Listing 12.1: The name *lemma1* was temporary and doesn't convey meaning

```
theory set
begin
 ...

public hitac intro := ⊆—def | ∩—def | id

public lemma inter_comm: A ∩ B ⊆ B ∩ A
proof( intro )
show x ∈ A ∩ B ⊢ x ∈ B ∩ A
    proof( intro )
      show x ∈ A ∩ B ⊢ x ∈ B
        by ∩—elim ; ax
      show x ∈ A ∩ B ⊢ x ∈ A
        by ∩—elim ; ax
    qed
qed

 ...
end
```

Listing 12.2: The name *inter_comm* conveys the fact this this is a proof of commutativity of intersection

as the preconditions are satisfied, will preserve the semantics of a theory. The transformation rule for renaming a lemma, as described in Section 10.3.5, is:

$$\frac{theory \xrightarrow{copyitem} theory' \qquad theory'' = theory'[lem\ lold := lem\ lnew]}{\begin{array}{l} thys \\ (\mathfrak{n}_{rename}, theory) \end{array} \xrightarrow{renamelemma'} \begin{array}{l} thys \\ (\mathfrak{n}_{rename}, theory'') \end{array}}$$

and is constructed by composing several, simpler refactorings together. To prove that this refactoring is semantics preserving, I guarantee that the preconditions for each of the atomic refactorings are satisfied.

POLAR: Our POLAR framework uses a language-independent graph meta-model to represent proof documents in a generic way. Proof languages can be connected to the framework by the specification of *abstraction* rules to create an abstract graph from the AST of a theory that only includes details relevant to a particular refactoring. Graph rewriting is used to enrich the meta-model with dependency information and to perform refactorings; finally, our transformation model allows us to regain the modified syntax.

Refactorings can be specified in a declarative way using rewrite rules. The *rename lemma* refactoring is expressed by the rewrite rule shown in Listing 12.3, as described in Section 11.6.1. The rewrite rule takes a graph node that corresponds to the lemma to rename and a new name and preforms the refactoring as long as the preconditions (represented within the rewrite rule) are satisfied.

12.3 FUTURE WORK

Following the structure of this thesis, I have separated the strands of further work that I would like to pursue into the following sections.

12.3.1 *Hiproofs and Hitac*

The Hiproof and Hitac formalisms are aimed at foundational research into languages for interactive proof. It has already been noted by Aspinall et al. (2010) that, before Hitac could be used 'for real', meta-variables in logical formulae would need to be modelled in hiproofs; and, the Hitac language would need to be extended to include, for example, higher order tacticals, and binding for goals and logical terms. Furthermore, stemming from work presented in this thesis, the notion of normal representations for hiproofs and a denotational account for the goal swapping construct is desirable.

This thesis also introduced well-formedness for tactics. This judgement can, without requiring evaluation, help provide information on whether a tactic is going to fail. I have only looked at a very simple well-formedness judgement, but we could possibly use tactic arities to provide further information on the possible behaviour of the tactic. The recursive nature of tactics, combined with the potential for alternate paths in evaluation means that an exact arity cannot be pinned to a tactic, it may be

```
rule PLRenameLabel(item:NamedItem, var newname:string)
{
  negative {
    defnode:NamedItem;
    :SearchContextAbove(item, newname, defnode);
  }
  negative {
    defnode:NamedItem;
    :SearchContextBelow(item, newname, defnode);
  }
  iterated {
    item <—:res— uref:Uref;
    modify {
      eval { uref.name = newname; }
    }
  }
  modify {
    eval { item = newname; }
  }
}
```

Listing 12.3: Rename item refactoring rewrite rule

possible to describe *ranges* of arities (where recursion may generate dynamic numbers of subgoals) or a choice for arities (where alternation may proceed in different directions). Thus, in general a tactic like $REPEAT(conjI)$ may have the arity (or *type*) of $REPEAT(conjI) : 1 \Rightarrow 1..$. If $REPEAT$ was forced to apply $conjI$ at least once, it would be $REPEAT(conjI) : 1 \Rightarrow 2..$. Furthermore, sometimes the output arity would be dependent on the input. For example, the tactic $ALL(id)$ could be given a dependent arity as follows: $ALL(id) : \forall\phi.\phi \Rightarrow \phi$, since the number of outputs will be exactly the number of inputs. It would be interesting to investigate a calculus for composing tactics and their types.

12.3.2 *Hiscript*

The Hiscript was designed as an experimental language for investigating hierarchy and proof language semantics. As such, it lacks some features of real-life languages, for example:

- The declarative aspect of the language is quite restrictive with respect to goal ordering. The **show** statements, for example must occur in the same order that the proof block introduction tactic generated them. A variation on the rule B-PRF-SHOW could allow any order:

$$\frac{g_1 \equiv [\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n] \qquad \langle g_1, rotate^{i}_{left} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, g_2 \rangle \qquad \langle [\gamma_i], prf \rangle \Downarrow_{\mathcal{E}} \langle s_1 \rangle \qquad \langle tail(g_2), stmts \rangle \Downarrow_{\mathcal{E}} \langle s_2 \rangle}{\langle g_1, \text{show } name\text{: } \gamma_i \ prf \ stmts \rangle \Downarrow_{\mathcal{E}} \langle s \ ; \ (([show \ name] \ s_1) \otimes s_2) \rangle} \text{(B-PRF-SHOW')}$$

by using the *rotate* tactic, which in turn uses the *swap* construct.

- It would also be interesting to investigate the notion of a *proof sketch*, as described by Wiedijk (2003). In a declarative proof language, this would correspond to providing a declarative proof language that allowed justifications to be elided. Similarly, languages such as Ωmega (and also eSSence) provide default automation that attempts to solve 'trivial' goals that have an explicit justification Dietrich (2011).
- I would also like to introduce new derivative language constructs. The declarative language for Coq, for example, has a *suffices to show* statement that can be used for proof blocks whose introduction tactic only introduces one subgoal.
- The theory language is particularly minimalist. At the very least, it should be extended to allow for axioms and definitions and abbreviations as well as the investigation of alternative theory inheritance mechanisms.

Another area of future work for Hiscript would be to provide an instantiation of the hiproof framework with a logical framework in which I can introduce language features that deal naturally with assumptions and bound variables; furthermore, such a logical instantiation would enable the formal investigation of logical module systems, such as Coq's *sections* and Isabelle's *locales* (Kammüller et al., 1999; The Coq development team, 2004). A promising approach to providing a generic logic-independent language could follow Schairer (2006), who used Institutions (Goguen and Burstall, 1992) as an abstract representation for a logic when defining structured transformations of a software specification.

The evaluation semantics for Hiscript theories is linear: similar to the traditional Proof General style evaluation of a theory, where each item is evaluated sequentially in a step-by-step manner (Aspinall et al., 2007). This, however, is only one possibility. In Haskell, for example, the order of definitions is not important and the compiler ensures that all dependencies are met before deciding upon an evaluation order. Furthermore, Wenzel's Isabelle/jEdit's asynchronous proof processing model offers an alternative approach that includes parallel proof checking. It would be interesting to attempt to capture such an interaction within this framework (Wenzel, 2012).

12.3.3 *eSSence*

Similarly to Hiscript, an obvious source of further work on eSSence is to extend my subset to cover more constructs and an instantiation of the hiproof framework with a more powerful type theory. The full pCIC of Coq, for instance would require the hiproof framework to model meta variables (Bertot and Castéran, 2004). Furthermore, the sophisticated matching that occurs in SSReflect would require extensions of the Hitac language.

Finally, another direction, I would like to attempt to separate the type theoretic constructs of SSReflect from the structuring components in order to provide a more natural translation of the SSReflect style of proof to other theorem proving systems. Recent work in this direction by Solovyev (2012) has ported a significant subset of the language to HOL Light. The implementation translates the SSReflect-style scripts to normal HOL Light tactics, mapping the standard SSReflect **apply** and **rewrite** tactics to the HOL Light equivalent.

12.3.4 *Hiscript refactoring*

The future work on refactoring Hiscript is closely related to progression of the language. The addition of more constructs to the language — a definitional framework, logical module system, a more sophisticated theory structure — introduces more possibilities for refactoring. Moreover, refactorings that may introduce gaps should also be investigated. A refactoring that introduces a new constructor for a datatype would introduce an extra case in the proofs of any properties involving that type.

Furthermore, the correctness proofs for refactorings can become unwieldy and it is future work to investigate both an extended representation of refactorings to include post-conditions and to consider a variant on the theory evaluation semantics to make proof document refactoring correctness more easily provable.

Many of the refactorings that I developed can be constructed by composing other, simpler refactorings. I currently compose refactorings in an ad-hoc manner and mostly sequentially as seen in, for example, the *rename item* refactoring. However, a refactoring such as *move item* is constructed using *swap items* many times until a condition is satisfied: that the item is in the desired place. It would be better to construct a simple language with a clear semantics for composing refactorings in a static and correct way. Furthermore, a formal notion of post-conditions would be useful for static composition.

12.3.5 *Polar*

The Polar refactoring framework offers a promising approach to generic proof language refactoring; however, the development is still in early stages and there are a number of important practical and theoretical directions that future work will take.

Besides expanding the number of implemented refactorings, we would like to expand the number of proof languages that are supported by Polar. In particular, we wish to implement a translation of the Isar proof language (Wenzel, 1999). Isar is used heavily in the Isabelle community and has many more constructs than Hiscript and ΩScript that we have currently connected. Furthermore, we already know that the meta-model is suitable for declarative and procedural proof languages, we would like to see if it holds tight for a language like SSReflect or even eSSence, which facilitates a very different type of proof style.

We would also like to include a dynamic connection to the theorem prover. This would allow us to attempt to close gaps introduced by refactorings such as *add a constructor*. Furthermore, we would like to establish a connection between the abstract

proof language and the resulting proof terms (e.g. to see whether a referenced label is indeed needed).

On the theoretical side, we would like to fully formalise the semantics of our bidirectional transformations and abstraction rules; in particular, we would like guarantees that, in certain situations, translation to and from the graph model is always possible. We would also like to investigate further constrains on the graph model that represent semantic restrictions on a proof language that are not covered by the *type graph*. A simple example of a constraint is the fact that in Hiscript all **have** statements must have a name: it cannot be blank. One possibility is to introduce the notion of illegal subgraphs. Finally, we would like to investigate how we can use our framework to guarantee correctness of refactorings. Since the refactorings happen in the meta-model, it is not as straightforward as the direct specification of refactorings in Hiscript. It would be also interesting to see how much of Polar could be expressed in a tool such as MetaEdit (Kelly et al., 1996).

On a grander scale, we plan to consider how our framework can be used as a means to automatically refactor a theory according to a specified style. We would also like to further investigate how we could use our graph meta- model for different applications. One possibility is to use it as a bridge between different proof languages allowing us to transform proofs in one language into another language.

12.4 CONCLUDING REMARKS

This thesis has introduced *proof refactoring* as a structured technique to make semantics preserving changes to the proof documents constructed by interactive theorem provers as part of a formal proof development. In order to study and understand what a refactoring is, Part 1 introduces the Hiscript framework: a tower of proof languages where each level is underpinned by a formal semantics. This formal semantics is then used in Part 2 to specify and prove correct over thirty semantics preserving transformations for the Hiscript language. Finally, Part 2 concludes with a description of Polar: a prototype framework for refactoring proof that I hope can be developed further to form a practical tool for proof engineering.

Part IV

APPENDIX

A

HISCRIPT PROPERTIES

This appendix details in full the various properties about the Hiscript language that have been omitted from the main body of the thesis. These properties are useful for the correctness proofs of refactorings in this thesis. In the next section, properties of the declarative proof language introduced in Chapter 4 are detailed. Section A.2.1 contains properties about the proof document language introduced in Chapter 5.

A.1 HISCRIPT PROOF PROPERTIES

A.1.1 *Minimal environments*

A minimal environment for proof evaluation is defined similarly to Hitac evaluation (as defined in Section 3.3.4). The slight subtlety in minimal environments for a proof *prf* is that some statements in a proof can extend the environment and these *local* definitions should not be considered a part of the minimal environment. Thus, they must be excluded from a minimal environment. The definition, however, is identical to that for tactics. There is a definition for a *prf* and for a list of statements *stmts* because it is often useful to know the minimal environment for the remaining list of statements in a block:

Definition 14 (Minimal environment (proof and statements))**.** $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$ is a minimal environment for a proof, *prf* (respectively, list of statements, *stmts*), if:

1. $(\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}}) \vdash \mathit{prf}$ (stmts);
2. For every proof environment $(\mathcal{T}', \mathcal{L}')$ such that $(\mathcal{T}', \mathcal{L}') \subset (\mathcal{T}_{\mathtt{min}}, \mathcal{L}_{\mathtt{min}})$, the proof (statements) are not well-formed.

Given an environment $(\mathcal{T}, \mathcal{L})$ and a proof *prf* that is well-formed under that environment, define the environment $(\mathcal{T}\big|_{\mathit{prf}}, \mathcal{L}\big|_{\mathit{prf}})$ as follows:

$$\begin{aligned} \mathcal{T}\big|_{\mathit{prf}} &= \{\ (n, \mathcal{T}(n))\ \mid\ n \in (\mathit{tacs}(\mathit{prf}) \setminus \mathit{localnames}(\mathit{prf}))\ \} \\ \mathcal{L}\big|_{\mathit{prf}} &= \{\ (n, \mathcal{L}(n))\ \mid\ n \in (\mathit{lemmas}(\mathit{prf}) \setminus \mathit{localnames}(\mathit{prf}))\ \} \end{aligned}$$

where *tacs* and *lemmas* are extensions to those defined inductively on the structure of hitac tactics in Section 3.3.4. The function *localnames* returns all the local definitions (tactics and lemmas) in a proof block. As expected, this turns out to be a minimal environment:

Theorem 44 (Minimality of $(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})$)**.** *Given an initial environment $(\mathcal{T}, \mathcal{L})$ and a proof prf, the environment $(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})$, defined above, is a minimal environment.*

Proof. The proof is similar to the hitac equivalent, except caution is needed when dealing with local definitions. □

A similar definition and theorem applies to the *stmt-restricted* environment, written $(\mathcal{T}|_{stmts}, \mathcal{L}|_{stmts})$:

Theorem 45 (Minimality of $(\mathcal{T}|_{stmts}, \mathcal{L}|_{stmts})$)**.** *Given an initial environment $(\mathcal{T}, \mathcal{L})$ and a list of statements $stmts$, the environment $(\mathcal{T}|_{stmts}, \mathcal{L}|_{stmts})$, defined above, is a minimal environment.*

If a proof evaluates under an environment $(\mathcal{T}, \mathcal{L})$, then it will certainly still evaluate under the minimal environment and also any well-formed environment in between:

Theorem 46 (Evaluation of proof under minimal environment)**.** *For a given proof environment $(\mathcal{T}, \mathcal{L})$, a proof prf, and a goal γ such that $\langle [\gamma], prf \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$, then:*

$$\langle [\gamma], prf \rangle \Downarrow_{(\mathcal{T}|_{prf}, \mathcal{L}|_{prf})} \langle s \rangle.$$

Furthermore, if $(\mathcal{T}', \mathcal{L}')$ is a well-formed proof environment such that

$$(\mathcal{T}|_{prf}, \mathcal{L}|_{prf}) \subseteq (\mathcal{T}', \mathcal{L}') \subseteq (\mathcal{T}, \mathcal{L})$$

then $\langle [\gamma], prf \rangle \Downarrow_{(\mathcal{T}',\mathcal{L}')} \langle s \rangle$.

Theorem 47 (Evaluation of statements under minimal environment)**.** *For a given proof environment $(\mathcal{T}, \mathcal{L})$, a list of statements $stmts$, and a list of goals g such that $\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$, then:*

$$\langle g, stmts \rangle \Downarrow_{(\mathcal{T}|_{stmts}, \mathcal{L}|_{stmts})} \langle s \rangle.$$

Furthermore, if $(\mathcal{T}', \mathcal{L}')$ is a well-formed proof environment such that

$$(\mathcal{T}|_{stmts}, \mathcal{L}|_{stmts}) \subseteq (\mathcal{T}', \mathcal{L}') \subseteq (\mathcal{T}, \mathcal{L})$$

then $\langle g, stmts \rangle \Downarrow_{(\mathcal{T}',\mathcal{L}')} \langle s \rangle$.

A.1.2 *Environment extension*

The general property of closure under environment extension proved in Theorem 8 does not extend to Hiscript proofs. To see why, consider the extended tactic environment with *newtac* added (where *newtac* doesn't already exist in the tactic environment):

$$\mathcal{T}' := \mathcal{T}[newtac \mapsto ([], id)]$$

and imagine the proof (which was well-formed under $(\mathcal{T}, \mathcal{L})$) is of the form shown in Listing 4.6. Now, it will fail well-formedness checking under the extended environment $(\mathcal{T}', \mathcal{L})$ since the name *newtac* will already exist in the environment: a failure of the precondition to the rule T-Prf-Tac. Thus, the extended environment must be restricted to exclude locally defined names.

```
proof
 ...

tac newtac := ...

 ...
qed
```

Listing A.1: A proof introducing a local tactic definition *newtac*

Theorem 48 (Closure of *prf* well-formedness under environment extension)**.** *Assume* $(\mathcal{T},\mathcal{L}) \subset (\mathcal{T}',\mathcal{L}')$ *and* $(\mathcal{T},\mathcal{L}) \vdash prf$*. If*

$$localnames(prf) \cap (names(\mathcal{T}') \cup names(\mathcal{L}')) = \{\}$$

then $(\mathcal{T}',\mathcal{L}') \vdash prf$*, where* ***localnames*** *is defined in the obvious inductive fashion on* prf*.*

Proof. By induction on the proof well-formedness judgement with appropriate calls to the equivalent theorem for tactic well-formedness, Theorem 8. □

A similar results holds for a list of statements:

Theorem 49 (Closure of statements typability under environment extension)**.** *Assume* $(\mathcal{T},\mathcal{L}) \subset (\mathcal{T}',\mathcal{L}')$ *and* $(\mathcal{T},\mathcal{L}) \vdash stmts$*. If*

$$localnames(stmts) \cap (names(\mathcal{T}') \cup names(\mathcal{L}')) = \{\}$$

then $(\mathcal{T}',\mathcal{L}') \vdash stmts$*.*

Furthermore, if for a given goal γ, a well-formed proof successfully evaluates under the environment $(\mathcal{T},\mathcal{L})$ then, extending the environment will preserve evaluation:

Theorem 50 (Closure of proof evaluation under environment extension)**.** *Assume* $(\mathcal{T},\mathcal{L}) \subset (\mathcal{T}',\mathcal{L}')$ *and for some* γ *that* $\langle \gamma, prf\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle$*. If*

$$localnames(prf) \cap (names(\mathcal{T}') \cup names(\mathcal{L}')) = \{\}$$

then $\langle [\gamma], prf\rangle \Downarrow_{(\mathcal{T}',\mathcal{L}')} \langle s\rangle$*.*

Again, a similar theorem holds for statements:

Theorem 51 (Closure of statement evaluation under environment extension)**.** *Assume* $(\mathcal{T},\mathcal{L}) \subset (\mathcal{T}',\mathcal{L}')$ *and for some list of goals* g *that* $\langle g, stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle$*. If*

$$localnames(stmts) \cap (names(\mathcal{T}') \cup names(\mathcal{L}')) = \{\}$$

then $\langle g, stmts\rangle \Downarrow_{(\mathcal{T}',\mathcal{L}')} \langle s\rangle$*.*

A.2 HISCRIPT THEORIES PROPERTIES

A.2.1 *Theory properties*

Just as it is possible to consider the minimal environment for an individual Hiscript proof or an individual hitac term, it is possible to consider a minimal environment for a whole theory. That is, the smallest imported environment to evaluate the whole theory. In analogy with excluding local definitions in proofs, both local definitions and *theory item definitions* are excluded in defining a minimal environment for a theory. This definition is expressed in terms of the theory items that a theory consists of:

Definition 15 (Minimal environment for theory items)**.** An imported environment $(\mathcal{T}_{min}, \mathcal{L}_{min})$ is a minimal environment for the theory items *thyitems* if

- $(\mathcal{T}_{min}, \mathcal{L}_{min}) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle$.
- For any well-formed environment $(\mathcal{T}_i, \mathcal{L}_i) \subset (\mathcal{T}_{min}, \mathcal{L}_{min})$, the theory items are not well-formed.

Assuming that an environment $(\mathcal{T}_i, \mathcal{L}_i)$ is suitable to judge a list of theory items *thyitems* well-formed and constructs the environment $(\mathcal{T}_{thyitems}, \mathcal{L}_{thyitems})$, then I define the *thyitems-restriction* $(\mathcal{T}_\mathfrak{i}\big|_{thyitems}, \mathcal{T}_\mathfrak{i}\big|_{thyitems})$ of $(\mathcal{T}_i, \mathcal{L}_i)$ as follows:

$$\mathcal{T}_\mathfrak{i}\big|_{thyitems} = \Big(\bigcup_{item \in thyitems} (\mathcal{T}_\mathfrak{i} \cup \mathcal{T}_{thyitems})\Big|_{item}\Big) \setminus \{(\mathfrak{n}, \mathcal{T}_{thyitems}(\mathfrak{n})) \mid \mathfrak{n} \in \mathit{tactics}(\mathit{thyitems})\}$$

$$\mathcal{L}_\mathfrak{i}\big|_{thyitems} = \Big(\bigcup_{item \in thyitems} (\mathcal{L}_\mathfrak{i} \cup \mathcal{L}_{thyitems})\Big|_{item}\Big) \setminus \{(\mathfrak{n}, \mathcal{L}_{thyitems}(\mathfrak{n})) \mid \mathfrak{n} \in \mathit{lemmas}(\mathit{thyitems})\}$$

That is, it is the union of all the minimal environments of all the containing items, with the tactics/lemmas defined in that list of theory items removed. I state, without proof, that this is indeed a minimal environment:

Theorem 52 (The *thyitems-restriction* is a minimal environment)**.** *Let* $(\mathcal{T}_i, \mathcal{L}_i)$ *and thyitems be such that*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle.$$

Then $(\mathcal{T}_\mathfrak{i}\big|_{thyitems}, \mathcal{L}_\mathfrak{i}\big|_{thyitems})$ *is well-formed and is a minimal environment for thyitems.*

If a list of theory items evaluates under an import environment $(\mathcal{T}_i, \mathcal{L}_i)$ then so to will the minimal environment and any other well-formed environment between them:

Theorem 53 (Evaluation of minimal environment for theory items)**.** *Let* $(\mathcal{T}_i, \mathcal{L}_i)$ *and thyitems be such that*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

Then

$$(\mathcal{T}_\mathfrak{i}\big|_{thyitems}, \mathcal{L}_\mathfrak{i}\big|_{thyitems}) \vdash \mathit{thyitems} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

Furthermore, let $(\mathcal{T}'_{\mathfrak{i}}\mathcal{L}'_{\mathfrak{i}})$ *be a well-formed environment map such that* $(\mathcal{T}_{\mathfrak{i}}\big|_{thyitems}, \mathcal{L}_{\mathfrak{i}}\big|_{thyitems}) \subseteq (\mathcal{T}'_{\mathfrak{i}}\mathcal{L}'_{\mathfrak{i}}) \subseteq (\mathcal{T}_i, \mathcal{L}_i)$. *Then*

$$(\mathcal{T}'_{\mathfrak{i}}\mathcal{L}'_{\mathfrak{i}}) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

It is also possible to add more to an imported environment for a list of theory items:

Theorem 54 (Environment extension for theory items)**.** *Let thyitems be such that, for an environment* $(\mathcal{T}_i, \mathcal{L}_i)$*:*

$$\mathcal{D} \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

Then, for any well-formed environment $(\mathcal{T}'_{\mathfrak{i}}, \mathcal{L}'_{\mathfrak{i}})$ *such that*

$$(\mathcal{T}_i, \mathcal{L}_i) \subseteq (\mathcal{T}'_{\mathfrak{i}}, \mathcal{L}'_{\mathfrak{i}}) \quad \textit{and} \quad names(thyitems) \cap (names(\mathcal{T}'_{\mathfrak{i}}) \cup names(\mathcal{L}'_{\mathfrak{i}})) = \varnothing$$

then $(\mathcal{T}'_{\mathfrak{i}}, \mathcal{L}'_{\mathfrak{i}}) \vdash thyitems \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$.

That is, one can extend an imported environment, so long as the extended environment does not contain any names that are introduced in the theory items (either as a lemma or tactic or as a *local* lemma or tactic inside a proof).

A.2.2 *Theory map properties*

Theories are evaluated in the context of a *theory map*. It is natural to consider both:

- The smallest theory map that will successfully evaluate the theory.
- How can the theory map be extended in a way that ensures it still evaluates.

I answer this first question with the following definition:

Definition 16 (Minimal theory map)**.** A theory map $\mathcal{D}_{min}$ is a *minimal theory map* for *theory* if:

1. $\mathcal{D}_{min} \vdash theory : \langle \mathcal{T}, \mathcal{L} \rangle$.
2. For every $\mathcal{D}' \subset \mathcal{D}_{min}$, *theory* is not well-formed.

Assuming that a theory map $\mathcal{D}$ is sufficient to judge *theory* well-formed, then the *theory-restriction* of $\mathcal{D}$ is defined as follows:

$$\mathcal{D}\big|_{theory} = \{(\mathfrak{n}, theory_{\mathfrak{n}}) \mid (\mathfrak{n}, theory_{\mathfrak{n}}) \in \mathcal{D} \ \wedge \ \mathfrak{n} \in imports(theory)\}$$

That is, the set of all theories from the theory map that are imported by *theory*. This forms a minimal theory map:

Theorem 55 (Theory restriction is minimal)**.** *Let* $\mathcal{D}$ *and theory be such that*

$$\mathcal{D} \vdash theory : \langle \mathcal{T}, \mathcal{L} \rangle.$$

Then $\mathcal{D}\big|_{theory}$ *is well-formed and is a minimal theory map for theory.*

I leave the definition of what it means to be well-formed for a theory map until the next section. Just as for declarative proofs, evaluation follows:

Theorem 56 (Evaluation of a theory under minimal theory map)**.** *Let $\mathcal{D}$ and theory be such that*

$$\mathcal{D} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

Then

$$\mathcal{D}\big|_{\mathit{theory}} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

Furthermore, let $\mathcal{D}'$ be a well-formed theory map such that $\mathcal{D}\big|_{\mathit{theory}} \subseteq \mathcal{D}' \subseteq \mathcal{D}$. Then

$$\mathcal{D}' \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

It is also possible to extend a theory map:

Theorem 57 (Closure under theory map extension)**.** *Let $\mathcal{D}$ and theory be such that*

$$\mathcal{D} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

Furthermore, let $\mathcal{D}'$ also be a well-formed theory map such that $\mathcal{D} \subseteq \mathcal{D}'$. Then,

$$\mathcal{D}' \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle.$$

B

SUMMARY OF THE HISCRIPT FRAMEWORK

This chapter summarises the Hiscript framework that Part 1 has constructed. In Section 3.2 on page 15 I started with a simple model of hierarchical proofs alongside a validation relation:

$$s \equiv ([m]\ \mathfrak{a}\ ;\ \mathfrak{b}\ \otimes\ id)\ ;\ [n]\ \mathfrak{c} \vdash \gamma_1 \longrightarrow []$$

which can be read as saying that s is a *valid proof* of the goal γ_1. The *atomic* tactics $\mathfrak{a}$, $\mathfrak{b}$, and $\mathfrak{c}$ are defined as follows:

$$\frac{\gamma_2 \quad \gamma_3}{\gamma_1}\ a \qquad \frac{}{\gamma_2}\ b \qquad \frac{}{\gamma_3}\ c$$

The linear syntax of hiproofs also has a graphical presentation, with the hiproof s looking like Figure B.1.

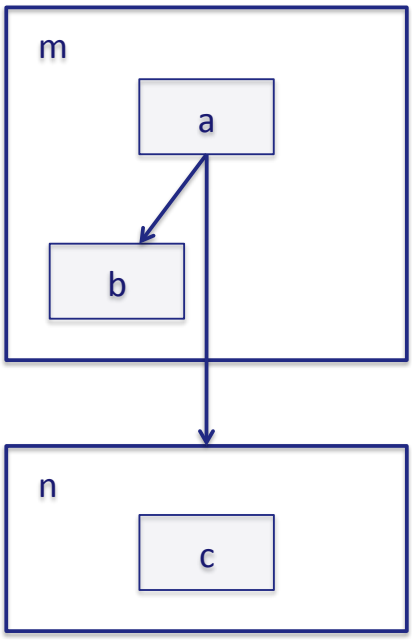

Figure B.1: The hiproof $([m]\ \mathfrak{a}\ ;\ \mathfrak{b}\ \otimes\ id)\ ;\ [n]\ \mathfrak{c}$

Now, the Hitac language, introduced in Section 3.3 on page 33, extends hiproofs with a number of additional constructs to facilitate proof search. In a neat way, any hiproof can be viewed as a tactic and hitacs are given meaning by an *evaluation* semantics that constructs hiproofs:

$$\langle \gamma_1, ([m]\ \mathfrak{a}\ ;\ \mathfrak{b}\ \otimes\ id)\ ;\ [n]\ \mathfrak{c} \rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, [] \rangle$$

Tactics, however, can be named and have parameters. For example, consider:

$$\mathfrak{m} := [\mathfrak{m}]\ \mathfrak{a}\ ;\ \mathfrak{b} \otimes \mathit{id}$$

$$\mathfrak{n} := [\mathfrak{n}]\ \mathfrak{c}$$

Furthermore, tactics can also refer to previously proved facts (or *lemmas*). Thus, tactic evaluation occurs within an *environment* of other tactics and lemmas that are available to be used. Thus, in an environment $(\mathcal{T}, \mathcal{L})$ containing the above two lemmas:

$$\langle \gamma_1, \mathfrak{m}\ ;\ \mathfrak{n} \rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, [] \rangle$$

On top of this *procedural* tactic language, I constructed the *declarative* proof language, Hiscript. Hiscript supports forward and backward proof steps and makes for human readable proofs. The following three proofs all solve the goal γ_1.

```
show γ1
proof(a)
  show γ2 by b
  show γ3 by c
qed
```

```
show γ1
proof(m)
  show γ3 by n
qed
```

```
show γ1
proof
 have *: γ3 by n
 from * show γ1 by m
qed
```

I give meaning to Hiscript proofs by means of an evaluation relation:

$$\langle \gamma_1, \mathit{prf} \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$$

that also constructs valid hiproofs. This evaluation relation does not return a list of remaining subgoals because in order to evaluate successfully a Hiscript proof must deal with all subgoals; however there is a way to leave goals unsolved, by the **gap** command. The syntax for Hiscript is given in Section 4.2 on page 46 and the semantics is given in Section 4.3 on page 49.

Finally, I address the issue of how to construct a *proof environment* $(\mathcal{T}, \mathcal{L})$ by introducing a notion of Hiscript *theory* and introduce a simple theory import mechanism. A theory like

```
theory example
import basics
begin

private hitac m := a ; ALL(TRY(b))
private hitac n  := c

public lemma example: γ1
  by m ; n

...
end
```

constructs an environment with the evaluation relation:

$$\mathcal{D} \vdash \mathit{theory} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$$

where $\mathcal{D}$ is a collection of theories and $(\mathcal{T}, \mathcal{L})$ is the constructed environment. The syntax for Hiscript theories is given in Section 5.3.1 on page 69 and the semantics is given in Section 5.3.2 on page 70.

C

PROOFS OF SEMANTICS PRESERVATION

This appendix contains the full correctness proofs for the refactorings that were specified in Chapters 9 and 10.

Each section details the proof of the refactorings. I have linked back to the specification of the refactoring for ease of reference.

C.1 DELETE UNUSED HAVE STATEMENT

The correctness of deleting a **have** statement, defined in Section 9.4 with name n from prf, parameterised by the goal γ and environment $(\mathcal{T}, \mathcal{L})$ is:

Theorem (Correctness of Delete Have)**.** *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$*, the preconditions hold, and* $prf \xrightarrow{deletehave} prf'$*, then* $\langle \gamma, prf' \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$*.*

Proof. The proof follows the modifier approach shown before. By assumption, the following derivation is valid:

$$\frac{\langle [\gamma_n], prf_n \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1 \rangle \qquad n \notin names(\mathcal{T} \cup \mathcal{L}) \qquad \langle g, stmts \rangle \Downarrow_{(\mathcal{T}, \mathcal{L}[n \longmapsto (\gamma_n,\ s_1)])} \langle s \rangle}{\langle g,\ \textbf{have}\ n\text{:}\ \gamma_n\ prf_n\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle}$$

Thus, we know that $\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s \rangle$ where $\mathcal{L}' = \mathcal{L}[n \longmapsto (\gamma_n,\ s_1)]$. We need to show that $\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$.

That is, it evaluates in the reduced environment

The approach is to use Theorem 47 on page 218, which states that if $(\mathcal{T}, \mathcal{L}')$ is a well-formed proof environment that evaluates successfully and

$$(\mathcal{T}\big|_{stmts}, \mathcal{L}\big|_{stmts}) \subseteq (\mathcal{T}, \mathcal{L}) \subseteq (\mathcal{T}, \mathcal{L}')$$

then $\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$.

Since $(\mathcal{T}, \mathcal{L}')$ is a indeed a well-formed environment and

$$(\mathcal{T}, \mathcal{L}) \subset (\mathcal{T}, \mathcal{L}')$$

we simply need to show that $(\mathcal{T}, \mathcal{L})$ is a superset of the minimal environment to have the result.

Now, by the precondition $n \notin lemmas(prf)$ and thus, $n \notin lemmas(stmts)$,so by definition $n \notin names(\mathcal{L}\big|_{stmts})$. The result follows. □

C.2 SWAP STATEMENTS

This section details the proof of the *swap statements* refactoring, defined in Section 9.11.

Theorem (Correctness of Swap Statements)**.** *If* $\langle \gamma, prf \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$*, the preconditions hold, and* $prf \xrightarrow{swapstmt} prf'$*, then* $\langle \gamma, prf' \rangle \Downarrow_{\mathcal{E}} \langle s \rangle$*.*

Proof. Correctness of this refactoring is argued using the same technique as above. In this case, I then need to perform a case analysis and show that each case of the modifier rule preserves semantics. I will show a couple of example cases:

- SS-Mod-HaveHave. On the left hand side of the rule, the evaluation is of the form:

$$\dfrac{\dfrac{\vdots}{\langle [\gamma_n], prf_n \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1 \rangle} \quad n \notin names(\mathcal{T} \cup \mathcal{L}) \qquad \dfrac{\dfrac{\vdots}{\langle [\gamma_m], prf_m \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_2 \rangle} \quad \dfrac{\vdots}{\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L}'')} \langle s' \rangle}}{\langle g, \textsf{have m: } \gamma_m\ prf_m\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s' \rangle}\ m \notin names(\mathcal{T} \cup \mathcal{L}')}{\langle g, \textsf{have n: } \gamma_n\ prf_n\ \textsf{have m: } \gamma_m\ prf_m\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle}$$

  where $\mathcal{L}' = \mathcal{L}[n \longmapsto (\gamma_n,\ s_1)]$ and $\mathcal{L}'' = \mathcal{L}'[m \longmapsto (\gamma_m,\ s_2)]$. We need to show that the derivation below is valid:

$$\dfrac{\dfrac{\vdots}{\langle [\gamma_m], prf_m \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2 \rangle} \quad m \notin names(\mathcal{T} \cup \mathcal{L}) \qquad \dfrac{\dfrac{\vdots}{\langle [\gamma_n], prf_n \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_1 \rangle} \quad \dfrac{\vdots}{\langle g, stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L}'')} \langle s' \rangle}}{\langle g, \textsf{have n: } \gamma_n\ prf_n\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s' \rangle}\ n \notin names(\mathcal{T} \cup \mathcal{L}')}{\langle g, \textsf{have m: } \gamma_m\ prf_m\ \textsf{have n: } \gamma_n\ prf_n\ stmts \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle}$$

  where this time $\mathcal{L}' = \mathcal{L}[m \longmapsto (\gamma_m,\ s_2)]$ and $\mathcal{L}'' = \mathcal{L}'[n \longmapsto (\gamma_n,\ s_1)]$.

  For the name freshness conditions $m \notin names(\mathcal{T} \cup \mathcal{L})$ and $n \notin names(\mathcal{T} \cup \mathcal{L}')$, noting that $n \neq m$, we use the assumptions from evaluation of the left hand side. We can show that $\langle [\gamma_m], prf_m \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2 \rangle$ is a valid evaluation with the help of the precondition for the transformation rule and an appeal to the evaluation under minimal environments (Theorem 46 on page 218). Similarly, to show $\langle [\gamma_n], prf_n \rangle \Downarrow_{(\mathcal{T},\mathcal{L}')} \langle s_1 \rangle$ we use closure under environment extension (Theorem 50 on page 219).

- SS-Mod-ShowApp. Similarly, the original evaluation at this point looks like:

$$\dfrac{\dfrac{\vdots}{\langle [\gamma], prf_n \rangle \Downarrow_{\mathcal{E}} \langle s_1 \rangle} \qquad \dfrac{\dfrac{\vdots}{\langle g, t \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_2, g' \rangle} \quad \dfrac{\vdots}{\langle g', stmts \rangle \Downarrow_{\mathcal{E}} \langle s_3 \rangle}}{\langle g, \textsf{apply } t\ stmts \rangle \Downarrow_{\mathcal{E}} \langle s_2\ ;\ s_3 \rangle}}{\langle \gamma :: g, \textsf{show n: } \gamma\ prf_n\ \textsf{apply } t\ stmts \rangle \Downarrow_{\mathcal{E}} \langle ([show\ n]\ s_1) \otimes (s_2\ ;\ s_3) \rangle}$$

and for the swapped statements, we need to show that the following is a valid derivation:

$$\dfrac{\dfrac{}{\langle \gamma, id\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle id, \gamma\rangle} \quad \dfrac{\vdots}{\langle g, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_2, g'\rangle}}{\langle \gamma :: g, id \otimes t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle id \otimes s_2, \gamma :: g'\rangle} \quad \dfrac{\dfrac{\vdots}{\langle [\gamma], prf_n\rangle \Downarrow_{\mathcal{E}} \langle s_1\rangle} \quad \dfrac{\vdots}{\langle g', stmts\rangle \Downarrow_{\mathcal{E}} \langle s_3\rangle}}{\langle \gamma :: g', \textsf{show n: } \gamma\ prf_n\ stmts\rangle \Downarrow_{\mathcal{E}} \langle ([show\ n]\ s_1) \otimes s_3\rangle}$$

$$\langle \gamma :: g, \textsf{apply } id \otimes t\ \textsf{show n: } \gamma\ prf_n\ stmts\rangle \Downarrow_{\mathcal{E}} \langle (id \otimes s_2)\ ;\ ([show\ n]\ s_1) \otimes s_3\rangle$$

which is simple, given the assumptions and appealing to the hitac evaluation rules B-Tac-Tens and B-Tac-Id.

□

C.3 BACKWARD STYLE PROOF TO FORWARDS STYLE PROOF

This section details the proof of the *backwards to forwards* proof refactoring, described in Section 9.12.

Theorem (Correctness of backward to forward refactoring)**.** *If* $\langle \gamma, prf\rangle \Downarrow_{\mathcal{E}} \langle s\rangle$*, the preconditions hold, and* $prf \xrightarrow{back2forward} prf'$*, then* $\langle \gamma, prf'\rangle \Downarrow_{\mathcal{E}} \langle s'\rangle$*.*

Proof. The proof proceeds by induction on the transformation rules. First note, that at the top level of transforming the proof, we have a valid derivation:

$$\dfrac{\dfrac{\vdots}{\langle \gamma, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, [\gamma_1, \ldots, \gamma_k]\rangle} \quad \dfrac{\vdots}{\langle [\gamma_1, \ldots, \gamma_k], stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}}{\langle \gamma, \textsf{proof}(t)\ stmts\ \textsf{qed}\rangle \Downarrow_{\mathcal{E}} \langle s_1\ ;\ s_2\rangle}$$

and need to show that the following is also a valid derivation (noting that a proof on its own is syntactic sugar for proof(id)):

$$\dfrac{\dfrac{}{\langle \gamma, id\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle id, \gamma\rangle} \quad \dfrac{\vdots}{\langle \gamma, stmts_{b2f}\ \textsf{from } n_1\ \ldots\ n_k \textsf{ show } n : \gamma \textsf{ by } t\rangle \Downarrow_{\mathcal{E}} \langle s\rangle}}{\langle \gamma, \textsf{proof}(id)\ stmts_{b2f}\ \textsf{from } n_1\ \ldots\ n_k \textsf{ show } n : \gamma \textsf{ by } t\ \textsf{qed}\rangle \Downarrow_{\mathcal{E}} \langle id\ ;\ s\rangle}$$

which requires us to show that

$$\langle \gamma, stmts_{b2f}\ \textsf{from } n_1\ \ldots\ n_k \textsf{ show } n : \gamma \textsf{ by } t\rangle \Downarrow_{\mathcal{E}} \langle s\rangle$$

evaluates successfully, where the following facts are available:

- $\langle \gamma, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, [\gamma_1, \ldots, \gamma_n]\rangle$;
- $\langle [\gamma_1, \ldots, \gamma_n], stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle$.

Induct on list of statements. The cases are, where $stmts = stmt :: stmts'$:

- $stmt = \textsf{have}\ lem : \gamma_{lem}\ prf_{lem}$. Then, we know the following is a valid derivation:

$$\frac{\begin{array}{c}\vdots\\ \langle \gamma_{lem}, prf_{lem}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_{lem}\rangle\end{array} \qquad \begin{array}{c}\vdots\\ \langle [\gamma_1,\ldots,\gamma_k], stmts'\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[lem\mapsto(\gamma_{lem},s_{lem})])} \langle s\rangle\end{array}}{\langle [\gamma_1,\ldots,\gamma_k], \textsf{have}\ lem : \gamma_{lem}\ prf_{lem}\ stmts'\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s\rangle}$$

  By the rule B2F-HAVE (where $stmts_{b2f} = stmt_{b2f} :: stmts'_{b2f}$), we know that $stmt_{b2f} = \textsf{have}\ lem : \gamma_{lem}\ prf_{lem}$ and we can show that the following derivation is valid by an appeal to the induction hypothesis (for the right branch) and using the assumption from above (for the left branch).

$$\frac{\begin{array}{c}\vdots\\ \langle \gamma_{lem}, prf_{lem}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_{lem}\rangle\end{array} \qquad \begin{array}{c}\vdots\\ \langle [\gamma], stmts'_{b2f}\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[lem\mapsto(\gamma_{lem},s_{lem})])} \langle s_{b2f}\rangle\end{array}}{\langle [\gamma], \textsf{have}\ lem : \gamma_{lem}\ prf_{lem}\ stmts'_{b2f}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_{b2f}\rangle}$$

  The transformation rules B2F-TAC and B2F-FROMHAVE have similar proofs.

- $stmt = \textsf{show}\ lem : \gamma_1\ prf_{lem}$. The following is a valid derivation:

$$\frac{\begin{array}{c}\vdots\\ \langle \gamma_1, prf_{lem}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle\end{array} \qquad \begin{array}{c}\vdots\\ \langle [\gamma_2,\ldots,\gamma_k], stmts'\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle\end{array}}{\langle [\gamma_1,\ldots,\gamma_k], \textsf{show}\ lem : \gamma_1\ prf_{lem}\ stmts'\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle ([show\ lem]\ s_1) \otimes s_2\rangle}$$

  Now, by the rule B2F-SHOW, we know (where $stmts_{b2f} = stmt_{b2f} :: stmts'_{b2f}$) that $stmt_{b2f} = \textsf{have}\ lem : \gamma_1\ prf_{lem}$ and we can show that the following derivation is valid by an appeal to the induction hypothesis (for the right branch) and using the assumption from above (for the left branch) and using the rule B-PRF-HAVE:

$$\frac{\begin{array}{c}\vdots\\ \langle \gamma_1, prf_{lem}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle\end{array} \qquad \begin{array}{c}\vdots\\ \langle [\gamma], stmts'_{b2f}\rangle \Downarrow_{(\mathcal{T},\mathcal{L}[lem\mapsto(\gamma_1,s_1)])} \langle s_{b2f}\rangle\end{array}}{\langle [\gamma], \textsf{have}\ lem : \gamma_{lem}\ prf_{lem}\ stmts'_{b2f}\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s_{b2f}\rangle}$$

  A similar argument holds for B2F-FROMSHOW.

- Finally, we deal with the base case where the list of statements is $stmts = []$. Here the evaluation rule is $\langle [], \rangle \Downarrow_{\mathcal{E}} \langle \langle\rangle\rangle$. This is refactored to

  $$\textsf{from}\ n_1\ \ldots\ n_k\ \textsf{show}\ \gamma\ \textsf{by}\ t$$

  We need to show the following derivation is valid:

$$\frac{\mathcal{L}(n_1) = (\gamma_1, s_1)\quad \ldots\quad \mathcal{L}(n_k) = (\gamma_k, s_k) \qquad \begin{array}{c}\vdots\\ \langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{(\mathcal{T},\mathcal{L})} \langle s, [\gamma_1,\ldots,\gamma_k]\rangle\end{array} \qquad \langle [], []\rangle \Downarrow_{\mathcal{E}} \langle\langle\rangle\rangle}{\langle [\gamma], \textsf{from}\ n_1\ \ldots\ n_k\ \textsf{show}\ \gamma\ \textsf{by}\ t\ \langle\rangle\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle (s\ ; (([n_1]\ s_1) \otimes \ldots \otimes ([n_k]\ s_k))) \otimes \langle\rangle\rangle}$$

Using the evaluation rule B-PRF-FROM1, we need to verify:

- $\langle [\gamma], t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s, [\gamma_1, \ldots, \gamma_k]\rangle (\mathcal{T}, \mathcal{L})$. This is easy to show as it was an assumption in the original derivation. Although, note that the environments are different, but closure under environment extension works here.
- For each n_i, we must have $\mathcal{L}(n_i) = (\gamma_i, s_i)$, which can be shown by tracking the extension of the environment through the derivation.

□

C.4 DECLARATIVE TO PROCEDURAL

This section details the correctness proof for the *declarative to procedural* refactoring, from Section 9.14.

Theorem. *Correctness of declarative to procedural If* $\langle \gamma, prf\rangle \Downarrow_{\mathcal{E}} \langle s\rangle$ *and* $prf \xrightarrow{dec2proc} t_{prf}$, *then* $\langle \gamma, t_{prf}\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s'\rangle$.

Proof. This proof is a simple induction on the transformation, utilising the knowledge that $\langle \gamma, prf\rangle \Downarrow_{\mathcal{E}} \langle s\rangle$. The cases are as follows:

- **proof**(t) *stmts* **qed**. The evaluation for this is the following tree:

$$\frac{\dfrac{\vdots}{\langle \gamma, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle s_1, g\rangle} \qquad \dfrac{\vdots}{\langle g, stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}}{\langle \gamma, \textbf{proof}(t)\ stmts\ \textbf{qed}\rangle \Downarrow_{\mathcal{E}} \langle s_1\ ;\ s_2\rangle}$$

  and the transformed tactic is $t\ ;\ t_{stmts}$; thus, by appealing to the tactic evaluation rule B-TAC-SEQ, We show that the following is a valid derivation:

$$\frac{\dfrac{\vdots}{\langle \gamma, t\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle g, s_1\rangle} \qquad \dfrac{\vdots}{\langle g, t_{stmts}\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s_2'\rangle}}{\langle \gamma, t\ ;\ t_{stmts}\rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s_1\ ;\ s_2'\rangle}$$

  which is trivial using the induction hypothesis and the original derivation tree.

- **show** n : γ *prf* *stmts*. This is evaluated with the following:

$$\frac{\dfrac{\vdots}{\langle \gamma, prf\rangle \Downarrow_{\mathcal{E}} \langle s_1\rangle} \qquad \dfrac{\vdots}{\langle g, stmts\rangle \Downarrow_{\mathcal{E}} \langle s_2\rangle}}{\langle \gamma :: g, \textbf{show}\ n :\ \gamma\ prf\ stmts\rangle \Downarrow_{\mathcal{E}} \langle ([show\ n]\ s_1)\ \otimes\ s_2\rangle}$$

  and, using the tactic evaluation rule B-TAC-TENS, we can show that the refactored tactic $t_{prf}\ \otimes\ t_{stmts}$ is also valid with two appeals to the induction hypothesis:

$$\frac{\vdots \qquad\qquad \vdots}{\langle \gamma, t_{prf} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s_1 \rangle \qquad \langle g, t_{stmts} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s_2 \rangle}{\langle \gamma :: g, t_{prf} \ \otimes \ t_{stmts} \rangle \Downarrow^{\mathbf{t}}_{\mathcal{E}} \langle [], s_1 \ \otimes \ s_2 \rangle}$$

- The two cases for apply and from ... show are similar.
- $[]$. The empty list of statements is evaluated by the rule B-PRF-EMPTY as:

  $$\langle [], [] \rangle \Downarrow_{\mathcal{E}} \langle \langle \rangle \rangle$$

  and it is easy to show that the refactored tactic $\langle\rangle$ evaluates successfully using the rule B-TAC-EMPTY.

□

From this point, it is trivial to show that by t_{prf} evaluates successfully.

C.5 PROCEDURAL TO DECLARATIVE

This section details the proof of the *procedural to declarative* refactoring from Section 9.15.

Theorem (Correctness of Hiproof to Hiscript). *For a hiproof s in TNF, if* $s \xrightarrow{proc2dec} prf$

$$\langle g, s \rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle g', s \rangle$$

then

$$\langle g, prf \rangle \Downarrow_{(\{\},\{\})} \langle s' \rangle$$

Proof. The proof is an induction on the transformation rules. The cases are as follows:

- P2D-ID. Here we know that $\langle \gamma, id \rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle \gamma, id \rangle$ and by a combination of the rules B-PRF-SHOW and B-PRF-GAP (with the B-Prf-Empty being used in the background, but it will not mentioned again), then we know that the refactoring show γ gap also evaluates.
- P2D-ATOMIC1. Similarly, this case is dealt with by B-PRF-SHOW and B-PRF-BLK.
- P2D-ATOMIC1. If $\langle \gamma, a \rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle [\gamma_1, \ldots, \gamma_n, a \rangle$ then we need to show that the following evaluates:

```
show γ
proof(a)
 show γ1
   gap
 ...
 show γn
   gap
qed
```

Now, using the rules B-PRF-SHOW and B-PRF-BLK, along with the assumption that the tactic a evaluates, we just need to show that the list of statements

$$[\textsf{show}\ \gamma_1\ \textsf{gap}, \ldots, \textsf{show}\ \gamma_n\ \textsf{gap}]$$

evaluates against the list of goals $[\gamma_1, \ldots, \gamma_n]$ which is simple.

- P2D-LAB. This case is simple, using the induction hypothesis and the proof evaluation rules B-PRF-SHOW and B-PRF-LAB.
- Similarly, P2D-SEQ, applied to $a\ ;\ s$ uses the induction hypothesis for the resulting statements from transforming s.
- P2D-TENS. This case is when we have multiple input goals. These are dealt with by multiple **show** statements in Hiscript. Thus, each s_i is recursively transformed and the induction hypothesis says that, for each s_i, if $\langle \gamma_i, s_i \rangle \Downarrow^{\mathbf{t}}_{(\{\},\{\})} \langle g_i, s_i \rangle$ then $\langle \gamma_i, prf_i \rangle \Downarrow_{(\{\},\{\})} \langle s'_i \rangle$. Using these hypotheses and n instances of B-PRF-SHOW, we obtain the result.
- P2D-LABSEQ. Finally, the more complex arrangement of a label in a TNF proof is shown correct in a similar way to the other cases.

□

C.6 FLATTEN SUBPROOF

This section details a sketch of the correctness proof of the *flatten subproof* refactoring, detailed in Section 9.16.

Theorem 58 (Correctness of flatten subproof)**.** *If* $\langle \gamma, prf_{out} \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ *and if*

$$prf_{out} \xrightarrow{flattensubproof} prf'_{out}$$

then $\langle \gamma, prf_{out'} \rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \langle s' \rangle$.

Proof. The proof follows a similar argument to all modifier proofs and consists of three parts:

1. Parse through prf_{out} until you find the statement $stmt_{in}$ to modify. The initial evaluation proceeds as per the original proof. Thus, at this point we have a (possibly extended by local definitions) proof environment $(\mathcal{T}', \mathcal{L}')$ and a list of goals $\gamma_{in} :: g$ that the remaining list of statements: $stmt_{in} :: stmts$ must solve.
2. We must then show that if $stmt_{in}$ evaluated successfully on the goal γ_{in} and is transformed by the rule FS-MOD to $stmts_{in}$, then:

   $$\langle \gamma_{in}, stmts_{in} \rangle \Downarrow_{(\mathcal{T}',\mathcal{L}')} \langle s_{in} \rangle$$

3. Finally, we must show that the rest of the statements *stmts* that were not modified, still evaluate successfully against the list of goals g under a (possibly) updated environment $(\mathcal{T}'', \mathcal{L}'')$, which results from evaluating $stmts_{in}$. This is the point where the precondition is used, alongside the closure under environment extension theorem.

□

C.7 COPY AN ITEM

This section gives the proof of the *copy item* refactoring from Section 10.2.3.

Theorem (Correctness of well-formedness checking for copy lemma)**.** *If, for a given imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$ *and theory items thyitems:*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{and} \quad thyitems \xrightarrow{copylemma} thyitems'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}, \mathcal{L}[lnew \mapsto (vis, \gamma, id)] \rangle.$$

That is, the refactored theory constructs identical environments, except that the lemma environment has been extended to include one more item.

Proof. We proceed by induction. The base case is for the rule CL-LEM1. By assumption, the following is a valid derivation:

$$\frac{\dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle} \qquad l \notin names(thyitems) \qquad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash prf}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \ \textsf{vis lemma}\ l : \gamma\ prf : \langle \mathcal{T}, \mathcal{L}[l \mapsto (vis, \gamma, id)] \rangle}$$

and we need to shown that the following is a valid derivation:

$$\frac{\dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \ \textsf{vis lemma}\ l : \gamma\ prf : \langle \mathcal{T}, \mathcal{L}' \rangle} \qquad lnew \notin names(thyitems) \qquad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}') \vdash prf}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \ \textsf{vis lemma}\ l : \gamma\ prf \ \textsf{vis lemma}\ lnew : \gamma\ prf : \langle \mathcal{T}, \mathcal{L}'[lnew \mapsto (vis, \gamma, id)] \rangle}$$

where $\mathcal{L}' = \mathcal{L}[l \mapsto (vis, \gamma, id)]$. The derivation

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems \ \textsf{vis lemma}\ l : \gamma\ prf : \langle \mathcal{T}, \mathcal{L}' \rangle}$$

is exactly the assumption, so this is valid. Furthermore, $lnew \notin names(thyitems)$ by the first precondition. Thus, we just need to show that:

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}') \vdash prf}$$

is valid. To do this, use Theorem 48 on page 219. This requires the local names of *prf* not to clash with the new proof environment:

$$localnames(prf) \cap (names(\mathcal{T}) \cup names(\mathcal{L}')) = \{\}$$

Now we know that

$$localnames(prf) \cap (names(\mathcal{T}) \cup names(\mathcal{L})) = \{\}$$

which means we just need to show that $\mathtt{l} \notin \mathit{localnames}(\mathit{prf})$), which is the third precondition.

The step cases CL-LEM2, CL-TAC, and CL-HITAC all have a similar proof. I show the case CL-TAC and CL-LEM2. The case CL-HITAC is identical to CL-TAC.

- CL-TAC. We know, by assumption that the following derivation is valid:

$$\frac{\dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle} \quad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash t} \quad \mathtt{n} \notin \mathit{names}(\mathit{thyitems}) \quad \mathit{vars}(t) \subseteq \overline{\mathtt{X}}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \mathit{vis}\ \mathtt{tac}\ \mathtt{n}(\overline{\mathtt{X}}) := t : \langle \mathcal{T}[\mathtt{n} \mapsto (\mathit{vis}, \overline{\mathtt{X}}, t)], \mathcal{L} \rangle}$$

  and we need to show that the following is valid:

$$\frac{\dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}' : \langle \mathcal{T}, \mathcal{L}' \rangle} \quad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}') \vdash t} \quad \mathtt{n} \notin \mathit{names}(\mathit{thyitems}') \quad \mathit{vars}(t) \subseteq \overline{\mathtt{X}}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}'\ \mathit{vis}\ \mathtt{tac}\ \mathtt{n}(\overline{\mathtt{X}}) := t : \langle \mathcal{T}[\mathtt{n} \mapsto (\mathit{vis}, \overline{\mathtt{X}}, t)], \mathcal{L}' \rangle}$$

  where $\mathcal{L}' = \mathcal{L}[\mathit{lnew} \mapsto (\mathit{vis}, \gamma, \mathit{id})]$. Now, $(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}' : \langle \mathcal{T}, \mathcal{L}' \rangle$ is valid by the induction hypothesis and $\mathit{vars}(t) \subseteq \overline{\mathtt{X}}$ by assumption. Furthermore, $\mathtt{n} \notin \mathit{names}(\mathit{thyitems}')$ as a result of the precondition. Thus we need to show that

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}') \vdash t}$$

  is a valid derivation, which is easy using the environment extension theorem for tactics in Theorem 8 on page 41.

- CL-LEM-2. This case follows a similar pattern to the previous. In this subproof we need to show that the following is a valid derivation:

$$\frac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}}$$

  In this case, we use the second precondition, which states that:

  $$\mathit{lnew} \notin \mathit{localnames}(\mathit{prf})$$

  which allows us to again use the environment extension theorem, Theorem 48.

□

C.8 LOCAL TO GLOBAL

This section gives the proof of the *local to global* refactoring from Section 10.2.5.

Theorem (Correctness of well-formedness checking for local to global)**.** *If, for a given imported environment* $(\mathcal{T}_i, \mathcal{L}_i)$ *and theory items thyitems*

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{and} \quad \mathit{thyitems} \xrightarrow{\mathit{localglobal}} \mathit{thyitems}'$$

then:

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}' : \langle \mathcal{T}[\mathit{tname} \mapsto (\textsf{private}, \overline{X}, t)], \mathcal{L} \rangle.$$

That is, the refactored theory constructs identical environments, except that the tactic environment has been extended to include the new global tactic.

Proof. The proof is an induction. The two cases are:

- LG-Mod: the base case. By assumption, we know that the following is a valid derivation:

$$\frac{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle \qquad \mathfrak{n} \notin \mathit{names}(\mathit{thyitems}) \qquad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \cup_L (\mathcal{T}, \mathcal{L}) \vdash \mathit{prf}}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \textsf{vis lemma}\ \mathit{lem} : \gamma\ \ \mathit{prf} : \langle \mathcal{T}, \mathcal{L}[\mathit{lem} \mapsto (\mathit{vis}, \gamma, \mathit{id})] \rangle}$$

  where $\mathit{prf} \equiv \textsf{proof}(t)\ \textsf{tac}\ \mathit{tname}(\overline{X})\ :=\ t\ \mathit{stmts}\ \textsf{qed}$. We then need to show that the derivations:

$$\frac{\dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle} \quad \mathit{vars}(t) \subseteq \overline{X} \quad \mathit{tname} \notin \mathit{names}(\mathit{thyitems}) \quad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \cup_L (\mathcal{T}', \mathcal{L}) \vdash t}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \textsf{private tac}\ \mathit{tname}(\overline{X})\ :=\ t : \langle \mathcal{T}', \mathcal{L} \rangle}$$

  which forms the first assumption of this derivation

$$\frac{\ldots \qquad \mathit{lem} \notin \mathit{names}(\mathit{thyitems}\ \textsf{tac}\ \mathit{tname} \ldots) \qquad \dfrac{\vdots}{(\mathcal{T}_i, \mathcal{L}_i)^{\mathit{pub}} \cup_L (\mathcal{T}', \mathcal{L}) \vdash \mathit{prf}'}}{(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems}\ \textsf{private tac}\ \mathit{tname}(x)\ :=\ t\ \ \textsf{vis lemma}\ \mathit{lem} : \gamma\ \mathit{prf}' : \langle \mathcal{T}', \mathcal{L}' \rangle}$$

  are both valid. Write $\mathcal{T}' = \mathcal{T}[\mathit{tname} \mapsto (\textsf{private}, \overline{X}, t)]$ and $\mathcal{L}' = \mathcal{L}[\mathit{lem} \mapsto (\mathit{vis}, \gamma, \mathit{id})]$. For the first derivation, which states that adding the new tactic is still a well-formed theory, we need to show:

  1. $(\mathcal{T}_i, \mathcal{L}_i) \vdash \mathit{thyitems} : \langle \mathcal{T}, \mathcal{L} \rangle$, which comes from the assumption above.
  2. $\mathit{vars}(t) \subseteq \overline{X}$. This is a consequence of the well-formedness of the tactic in its original position.
  3. $\mathit{tname} \notin \mathit{names}(\mathit{thyitems})$. This is also a consequence of well-formedness of the tactic in its original position.

4. $(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L}) \vdash t$. Another consequence of well-formedness in the original position as exactly the same environment is passed to prf.

Then, for the second derivation, we need to show:

1. $lem \notin names(thyitems\ \textsf{tac}\ tname \ldots)$. I know from the original derivation that $lem \notin names(thyitems)$ and the fact that $tname \neq lem$ is a precondition to the refactoring.
2. Finally, we need to show that $(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}', \mathcal{L}) \vdash prf'$. By assumption, the following derivation is valid, writing $(\mathcal{T}_{prf}, \mathcal{L}_{prf})$ to abbreviate the combination of imported environment and previously constructed environment $(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}, \mathcal{L})$:

$$\dfrac{\dfrac{\vdots}{(\mathcal{T}_{prf}, \mathcal{L}_{prf}) \vdash t} \quad \dfrac{\dfrac{}{(\mathcal{T}_{prf}, \mathcal{L}_{prf}) \vdash \textsf{tac}\ tname(\overline{X})\ :=\ t : (\mathcal{T}_{prftac}, \mathcal{L}_{prf})} \quad \dfrac{\vdots}{(\mathcal{T}_{prftac}, \mathcal{L}_{prf}) \vdash stmts}}{(\mathcal{T}_{prf}, \mathcal{L}_{prf}) \vdash \textsf{tac}\ tname(\overline{X})\ :=\ t\ stmts}}{(\mathcal{T}_{prf}, \mathcal{L}_{prf}) \vdash \textsf{proof}(t)\ \textsf{tac}\ tname(\overline{X})\ :=\ t\ stmts\ \textsf{qed}}$$

where $\mathcal{T}_{prftac} = \mathcal{T}_{prf}[tname \mapsto (vis, \overline{X}, t)]$ and then need to show that the following refactored prf' is valid:

$$\dfrac{\dfrac{\vdots}{(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) \vdash t} \quad \dfrac{\vdots}{(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) \vdash stmts}}{(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) \vdash \textsf{proof}(t)\ stmts\ \textsf{qed}}$$

where $(\mathcal{T}_{prf'}, \mathcal{L}_{prf'})$ is a contraction of $(\mathcal{T}_i, \mathcal{L}_i)^{pub} \cup_L (\mathcal{T}', \mathcal{L})$. Now, from above, we know that $\mathcal{L}_{prf'} = \mathcal{L}_{prf}$ and that $\mathcal{T}_{prf'} = \mathcal{T}_{prf}[tname \mapsto (\textsf{private}, \overline{X}, t)]$. Thus, we know that:

- $(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) \vdash t$, by the environment extension theorem.
- and $(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) \vdash stmts$ by the observation that, from the original derivation, $(\mathcal{T}_{prf'}, \mathcal{L}_{prf'}) = (\mathcal{T}_{prftac}, \mathcal{L}_{prf})$, which means we can directly use the assumption.

- I now show the step case for the rule LG-ThyItem. The inductive hypothesis states that if

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems : \langle \mathcal{T}, \mathcal{L} \rangle$$

then

$$(\mathcal{T}_i, \mathcal{L}_i) \vdash thyitems' : \langle \mathcal{T}[tname \mapsto (\textsf{private}, \overline{X}, t)], \mathcal{L} \rangle.$$

so we just need to show that the individual theory item $thyitem$, which is not modified by the rule, is well-formed with respect to this extended environment. This is a direct consequence of the environment extension theorem for tactics and lemmas, depending on what type of theory item it is.

□

BIBLIOGRAPHY

Alkassar, E., Hillebrand, M. A., Leinenbach, D. C., Schirmer, N. W., Starostin, A., and Tsyban, A. (2009). Balancing the load. *Journal of Automated Reasoning*, 42(2-4):389–454. (Cited on page 116.)

Archive of Formal Proofs (2012). The archive of formal proofs. (Cited on page 8.)

Arias, T. B. C., van der Spek, P., and Avgeriou, P. (2011). A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586. (Cited on page 196.)

Aschbacher, M. (2005). Highly complex proofs and implications of such proofs. *Philosophical Transactions of The Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:2401–2406. (Cited on page 102.)

Aspinall, D., Denney, E., and Lüth, C. (2010). Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330. (Cited on pages 11, 15, 33, 41, and 211.)

Aspinall, D., Lüth, C., and Winterstein, D. (2007). A framework for interactive proof. In Kauers, M., Kerber, M., Miner, R., and Windsteiger, W., editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 161–175. Springer. (Cited on page 213.)

Bancilhon, F. and Spyratos, N. (1981). Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575. (Cited on page 196.)

Barendregt, H. (1992). Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press. (Cited on page 31.)

Bell Canada (2000). DATRIX abstract semantic graph reference manual (version 1.4). Technical report. (Cited on page 206.)

Bertot, Y. and Castéran, P. (2004). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer. (Cited on page 213.)

Boger, M., Sturm, T., and Fragemann, P. (2002). Refactoring browser for UML. In Aksit, M., Mezini, M., and Unland, R., editors, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*, pages 366–377. Springer. (Cited on page 110.)

Borba, P. and Sampaio, A. (2000). Basic laws of ROOL: an object-oriented language. *RITA*, 7(1):49–68. (Cited on page 106.)

Bourke, T., Daum, M., Klein, G., and Kolanski, R. (2012). Challenges and experiences in managing large-scale proofs. In Jeuring et al. (2012), pages 32–48. (Cited on pages 102 and 116.)

Butler, M. and Hallerstede, S. (2007). The Rodin formal modelling tool. In *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry*. (Cited on page 110.)

Chen, H. and Liao, H. (2010). A comparative study of view update problem. In *Data Storage and Data Engineering (DSDE)*, pages 83–89. (Cited on page 183.)

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. (1999). *Maude: Specification and Programming in Rewriting Logic*. SRI International. (Cited on page 106.)

Coen, C. S. (2010). Declarative representation of proof terms. *Journal of Automated Reasoning*, 44(1-2):25–52. (Cited on page 13.)

Corbineau, P. (2008). A declarative language for the coq proof assistant. In *Proceedings of the 2007 international conference on Types for proofs and programs*, TYPES'07, pages 69–84, Berlin, Heidelberg. Springer-Verlag. (Cited on page 12.)

Cornélio, M., Cavalcanti, A., and Sampaio, A. (2002). Refactoring by transformation. *Electronic Notes in Theoretical Computer Science*, 70(3):311 – 330. REFINE 2002, The BCS FACS Refinement Workshop (Satellite Eventof FLoC 2002). (Cited on pages 106 and 179.)

de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2007). Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163. (Cited on page 189.)

Denney, E., Power, J., and Tourlas, K. (2006). Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359. (Cited on pages 1, 10, 11, and 18.)

Dietrich, D. (2011). *Assertion Level Proof Planning with Compiled Strategies*. PhD thesis, Saarland University. (Cited on pages 12, 184, and 213.)

Eclipse (2012). Eclipse Java development tools. http://www.eclipse.org/jdt. (Cited on page 108.)

Eetvelde, N. V. and Janssens, D. (2003). A hierarchical program representation for refactoring. *Electronic Notes in Theoretical Computer Science*, 82(7):91 – 104. UNIGRA'03, Uniform Approaches to Graphical Process Specification Techniques (Satellite Event for ETAPS 2003). (Cited on page 107.)

Ettinger, R. and Verbaere, M. (2005). Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio. http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports. (Cited on page 109.)

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349. (Cited on page 196.)

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on pages 4, 105, 110, and 111.)

Garrido, A. and Meseguer, J. (2006). Formal specification and verification of Java refactorings. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 165–174, Washington, DC, USA. IEEE Computer Society. (Cited on page 106.)

Geiß, R., Batz, G., Grund, D., Hack, S., and Szalkowski, A. (2006). GrGen: A fast SPO-based graph rewriting tool. In *Third International Conference on Graph Transformation (ICGT 2006)*. Springer-Verlag, LNCS 4178. (Cited on page 198.)

Goguen, J. A. and Burstall, R. M. (1992). Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146. (Cited on page 213.)

Gonthier, G. (2008). The Four Colour Theorem: Engineering of a formal proof. *Computer Mathematics: 8th Asian Symposium, ASCM 2007*, pages 333–333. (Cited on pages 81 and 102.)

Gonthier, G. (2012). personal communication. (Cited on pages 83 and 113.)

Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., and Théry, L. (2007). A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA. (Cited on page 102.)

Gonthier, G., Mahboubi, A., and Tassi, E. (2008). A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA. (Cited on pages 13, 81, 83, 87, and 89.)

Gonthier, G. and Roux, S. L. (2009). An SSReflect Tutorial. Technical Report RT-0367, INRIA. (Cited on page 89.)

Gordon, M., Milner, R., Morris, L., Newey, M., and Wadsworth, C. (1978). A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 119–130, New York, NY, USA. ACM. (Cited on pages 8 and 41.)

Hales, T. C. (2006). Introduction to the Flyspeck project. In Coquand, T., Lombardi, H., and Roy, M.-F., editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. <http://drops.dagstuhl.de/opus/volltexte/2006/432> [date of citation: 2006-01-01]. (Cited on page 102.)

Harrison, J. (1996). Proof style. In Giménez, E. and Paulin-Mohring, C., editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France. Springer-Verlag. (Cited on page 8.)

Harrison, J. (1998). The HOL Light manual (1.0). (Cited on pages 11 and 59.)

Hofmann, M., Pierce, B., and Wagner, D. (2012). Edit lenses. In *ACM SIGPLAN Notices*, volume 47, pages 495–508. ACM. (Cited on page 196.)

Jeuring, J., Campbell, J. A., Carette, J., Reis, G. D., Sojka, P., Wenzel, M., and Sorge, V., editors (2012). *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*. Springer. (Cited on pages 236 and 241.)

Kammüller, F., Wenzel, M., and Paulson, L. C. (1999). Locales - a sectioning concept for Isabelle. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., and Théry, L., editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer. (Cited on page 213.)

Kelly, S., Lyytinen, K., and Rossi, M. (1996). Metaedit+: A fully configurable multi-user and multi-tool case and came environment. In *Proceedings of the 8th International Conference on Advances Information System Engineering*, CAiSE ;96, pages 1–21, London, UK, UK. Springer-Verlag. (Cited on page 215.)

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM. (Cited on pages 102 and 116.)

Knies, R. (2012). Theorem proof gains acclaim. http://research.microsoft.com/en-us/news/features/gonthierproof-101112.aspx. (Cited on page 102.)

Lakatos, I., Worrall, J., and Zahar, E. (1976). *Proofs and refutations: The logic of mathematical discovery*. Cambridge University press. (Cited on page 8.)

Lämmel, R. (2002). Towards generic refactoring. In *Proc. ACM SIGPLAN workshop on Rule-based programming (RULE)*, pages 15–28. ACM Press: New York NY. (Cited on pages 109 and 206.)

Lämmel, R. and Visser, J. (2003). A Strafunski application letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 357–375, London, UK, UK. Springer-Verlag. (Cited on page 108.)

Lamport, L. (1993). How to write a proof. *American Mathematical Monthly*, 102:600–608. (Cited on page 9.)

Li, H. (2006). *Refactoring Haskell Programs*. PhD thesis, Canterbury, Kent, UK. (Cited on page 107.)

Li, H. and Thompson, S. (2005). Formalisation of Haskell Refactorings. In van Eekelen, M. and Hammond, K., editors, *Trends in Functional Programming*. (Cited on pages 108 and 179.)

Li, H., Thompson, S., and Reinke, C. (2005). The Haskell Refactorer: HaRe, and its API. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science, http://www.sciencedirect.com/science/journal/15710661. (Cited on page 108.)

The Coq development team (2004). *The Coq proof assistant reference manual*. LogiCal Project. Version 8.0. (Cited on pages 13, 90, and 213.)

Mens, T., Eetvelde, N. V., Demeyer, S., and Janssens, D. (2005). Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276. (Cited on pages 106, 107, 179, 204, and 206.)

Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139. (Cited on page 104.)

Mens, T., Tourwé, T., and Muñoz, F. (2003). Beyond the Refactoring Browser: advanced tool support for software refactoring. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 39, Washington, DC, USA. IEEE Computer Society. (Cited on page 109.)

Milner, R., Tofte, M., and Harper, R. (1990). *Definition of standard ML*. MIT Press. (Cited on page 8.)

Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA. IEEE Computer Society. (Cited on page 102.)

NetBeans (2012). Netbeans - refactoring. http://refactoring.netbeans.org/. (Cited on page 108.)

Newman, M. H. A. (1942). On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2):223–243. (Cited on page 27.)

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer. (Cited on pages 12 and 59.)

Nozal, C. L., Sánchez, R. M., Crespo, Y., and Pérez, F. J. (2006). Towards a language independent refactoring framework. In Filipe, J., Shishkov, B., and Helfert, M., editors, *ICSOFT (1)*, pages 165–170. INSTICC Press. (Cited on page 206.)

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA. (Cited on pages 1, 100, 104, and 179.)

Peyton Jones, S. et al. (2003). The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1):0–255. (Cited on pages 100 and 107.)

Pons, O., Bertot, Y., and Rideau, L. (1998). Notions of dependency in proof assistants. In *User Interfaces for Theorem Provers, UITP*. (Cited on pages 104 and 116.)

Refactory Inc (2012). The refactoring browser. http://www.refactory.com/RefactoringBrowser/. (Cited on page 108.)

Roberts, D., Brant, J., and Johnson, R. E. (1997). A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263. (Cited on pages 108 and 109.)

Roberts, D. B. (1999). Practical analysis for refactoring. Technical report, Champaign, IL, USA. (Cited on pages 105 and 179.)

Rudnicki, P. (1992). An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330. (Cited on page 12.)

Schaefer, M. and de Moor, O. (2010). Specifying and implementing refactorings. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 286–301, New York, NY, USA. ACM. (Cited on page 181.)

Schairer, A. (2006). *Transformations of specifications and proofs to support an evolutionary formal software development*. PhD thesis. (Cited on page 213.)

Solovyev, A. (2012). *SSReflect/HOL Light*. Flyspeck. (Cited on page 214.)

Stepney, S., Polack, F., and Toyn, I. (2002). Refactoring in maintenance and development of Z specifications. *Electr. Notes Theor. Comput. Sci.*, 70(3). (Cited on page 109.)

Stevens, P. (2008). A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*, pages 408–424. Springer. (Cited on page 196.)

Sultana, N. and Thompson, S. (2008). Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*. ACM SIGPLAN. (Cited on page 108.)

Sunyé, G., Pollet, D., Traon, Y. L., and Jézéquel, J. M. (2001). Refactoring UML models. pages 134–148. (Cited on page 110.)

Tichelaar, S., Ducasse, S., and Demeyer, S. (2000). FAMIX and XMI. In *WCRE*, pages 296–298. (Cited on page 206.)

Verbaere, M., Ettinger, R., and de Moor, O. (2006). JunGL: a scripting language for refactoring. In Rombach, D. and Soffa, M. L., editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA. ACM Press. (Cited on pages 109 and 206.)

Wenzel, M. (1999). Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pages 167–184. (Cited on pages 12, 58, and 214.)

Wenzel, M. (2012). Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. *Electr. Notes Theor. Comput. Sci.*, 285:101–114. (Cited on page 213.)

Whitehead, A. N. and Russell, B. (1912). *Principia mathematica*, volume 2. University Press. (Cited on page 8.)

Whiteside, I., Aspinall, D., Dixon, L., and Grov, G. (2011). Towards formal proof script refactoring. In Davenport, J. H., Farmer, W. M., Urban, J., and Rabe, F., editors, *Calculemus/MKM*, volume 6824 of *Lecture Notes in Computer Science*, pages 260–275. Springer. (Cited on pages 45 and 60.)

Whiteside, I., Aspinall, D., and Grov, G. (2012). An essence of SSReflect. In Jeuring et al. (2012), pages 186–201. (Cited on page 81.)

Wiedijk, F. (2003). Formal proof sketches. In Berardi, S., Coppo, M., and Damiani, F., editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 378–393. Springer. (Cited on page 213.)

Wiedijk, F., editor (2006). *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer. (Cited on page 11.)

Wiedijk, F. (2012). A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1). (Cited on pages 12 and 58.)

Wiles, A. (1995). Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, 141(3):pp. 443–551. (Cited on page 8.)

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of September 11, 2017 (`classicthesis`).